Escola d'Enginyeria de Telecomunicació i Aeroespacial de Castelldefels

UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

BACHELOR'S DEGREE PROJECT

Title: Development of a Personal Area Network for biomedical measurements for Internet of Things (IoT)

Bachelor's degree: Bachelor's degree in Telecommunication Systems Engineering.

Author: Alex González i Juclà

Advisor: José Polo Cantero

Date: October, 30th 2018

Abstract

Internet of Things is a set of ever growing technologies and specialized devices that are increasingly influential in our everyday lives. IoT is all about connecting the physical and the digital worlds in one enabling the collection of real world data and the automation of processes. IoT turns your typical device into an smart, programmable one, more capable of interacting with humans and thus enabling users to better understand their surroundings through the data collected. This data collected by the IoT devices can then be used on all kinds of contemporary services and applications.

This project aims to implement an IoT application for biomedical measurements, consisting of a WSN(Wireless Sensor Network), where three sensor nodes will collect physical world measurements. This collected information will be transmitted to a routing device, that further send the information to the internet, where the user will be able to access the data in real time through a web browser and schedule some events.

In order to carry out the described scenario, a Raspberry Pi and four Zolertia Z1, three working as sensor nodes and one working as a routing node were used. The Z1 mote is powered by a low power MSP430 class microcontroller. Contiki was the operating system chosen to run the sensor nodes. In this scenario, Raspberry Pi plays the role of a router, enabling the connection of the WSN network and the internet.

To send the information from the nodes, a high-speed program was developed, aiming to beat the default restrictions that Contiki OS imposes on high-speed networks. The transport protocol chosen is UDP. On the receiving end, an UDP server and a python script were developed with the intent to send the collected data to our ASP.NET web server and mySQL database.

Finally connectivity tests and network speed tests of the deployed system are presented.

CONTENTS

LIST OF FIGURES

LIST OF TABLES AND SNIPPETS

1. INTRODUCTION

From its creation, the Internet has revolutionized various areas of society by completely changing the world. The first major change came with the World Wide Web that allowed access to information regardless of geographical location, making knowledge accessible globally. Then, the next revolution was mobile Internet that allowed the access to Internet from anywhere at anytime. Now a new revolution is present: the Internet of Things.

The IoT's major significant trend in recent years is the explosive growth of devices connected and controlled by the Internet. The wide range of applications for IoT technology mean that the specifics can be very different from one device to the next but there are basic characteristics shared by most.

IoT creates opportunities for more direct integration of the physical world into computer-based systems, resulting in efficiency improvements, economic benefits, and reduced human exertions.

The number of IoT devices increased 31% year-over-year to 8.4 billion in the year 2017 and it is estimated that there will be 30 billion devices by 2020. The global market value of IoT is projected to reach $7.1 trillion by 2020. [Fig 1.1.] provides a graph where the exponential increase in IoT adoption can be seen.

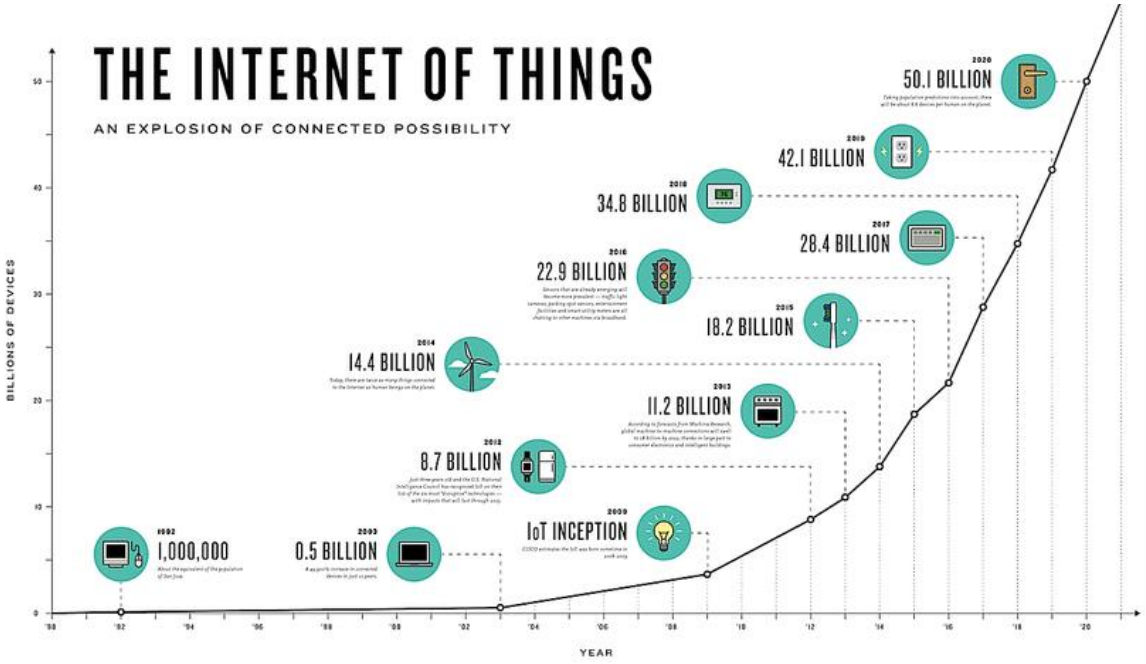

Fig 1.1. Adoption of IoT technology. [24]

In this context this project present implementation of an IoT application, composed of a Wireless Sensor Network (WSN) where sensor nodes collect data of temperature, battery's voltage and acceleration and send them to a central entity to be stored and processed.

The network is intended to be used in the future for biomedical purposes, and to do so, a minimum sampling rate has to be achieved.

1.1. Project Overview

This paper is organized as follows. Chapter 1 is a brief introduction to the project, its goals and its challenges. Chapter 2 presents the key concepts and breaks down the basic theory needed to properly comprehend the project. Chapter 3 focuses on the solution of the problem to be solved. Chapter 4 goes in detail to the results obtained from both the software developed and network implemented. And in the last chapter the conclusions are established and what remains open to future developments.

1.1.1 List of devices used

In the making of this project, a large amount of physical equipment and devices was required, here’s a list of all the devices and peripherals used.

- 4 Zolertia Z1
- Raspberry Pi - Model B
- 5 USB - Micro USB cables
- External battery charger
- Mobile phone charger adapter
- Ethernet Cat6 cable with RJ-45 connections
- Mouse, keyboard and an HDMI screen with its cable
- Computer with IIS, ASP4.5 and Linux capabilities

1.2. Project goals

The main goal of this project is the development of a network able to capture, measure and store biological signals, the measurement of this signals might be challenging, since a minimum sampling frequency will be required to do so, this minimum sampling frequency required will be further discussed in chapter [1.3].

The need of such high frequency will be especially demanding on our network, since most WSN’s are not prepared for high-speed environments, software

modifications will have to be made, optimizations will be mandatory and tough design decisions will have to be made.

The first step toward the stated goal will be to find our softwares of choice limitations regarding speed and trying to find a way around that.

Once every step of the software chain is properly configured to work at its maximum frequency an in-deep examination of software performance will be made to make sure that the code working on every device of the network is fully optimized for the needs of the project.

Finally, a user interface will be required to allow access to the biological signals measured. To allow proper user experience and interaction with the fast-paced data collected, a real-time data-visualization interface will be developed.

1.3. Biosensors and biomedical signals

A biosensor is any piece of hardware that interacts with a biological or physiological system to acquire a signal for either diagnostic or therapeutic purposes. Data gathered using biosensors are then processed using biomedical signal processing techniques as a first step toward facilitating human or automated interpretation.

This data gathered by the biosensors are biomedical signals, very low-power, noisy signals that will require further signal processing. These signals vary greatly in terms of speed and noise, on this project we will only take into account the signals provided in [Table 1.1.].

Temperature	32 – 42 °C	dc – 0.1 Hz
Phonocardiogram	80 dB above 100 µPa	5 Hz – 2 kHz
Respiratory rate	2 –50 breaths/min	0.1 – 10 Hz
Blood pH	6.8 – 7.8	dc – 2 Hz
Gastric pH	3 – 13	dc – 1 Hz
GSR	1 – 500 kΩ	0.01 – 1 Hz

Table 1.1. Biomedical measurements and its bandwidths.

As shown above, the most restrictive signal we would need to measure is a phonocardiogram, ranging from 5 Hz to 2 KHz of bandwidth.

Using the Nyquist sampling theorem, which states that the minimum sampling frequency of a system equals to double the highest frequency to be sampled:

$$Fs = 2 * fh$$

We get to the conclusion that:

$$Fs = 2 * 2KHz = 4KHz$$

In order to attain a decent enough Phonocardiogram, a minimum 4KHz sampling frequency would be required. This being attainable with the selected development environment and softwares will be further discussed in chapter [3].

2. IOT OVERVIEW

2.1. Internet of things

The internet of things(IoT), is a system of interrelated computing devices, mechanical and digital machines, objects, animals or people that are provided with unique identifiers and the ability to transfer data over a network without requiring human-to-human or human-to-computer interaction. The current paradigm of IoT attempts to digitize everything that surrounds us, always with the purpose of acquiring data from the real world and translating it to an improvement the quality of life. To do so, a plethora of different information and communication technologies(ICTs) have been developed in the recent years, and continue to be developed.

IoT adds a new dimension to ICTs, which is providing of ICTs capabilities to any simple device imaginable without the need of complex computers or massive bandwidth. This new dimension and its properties are further explained in [Fig 2.1.].

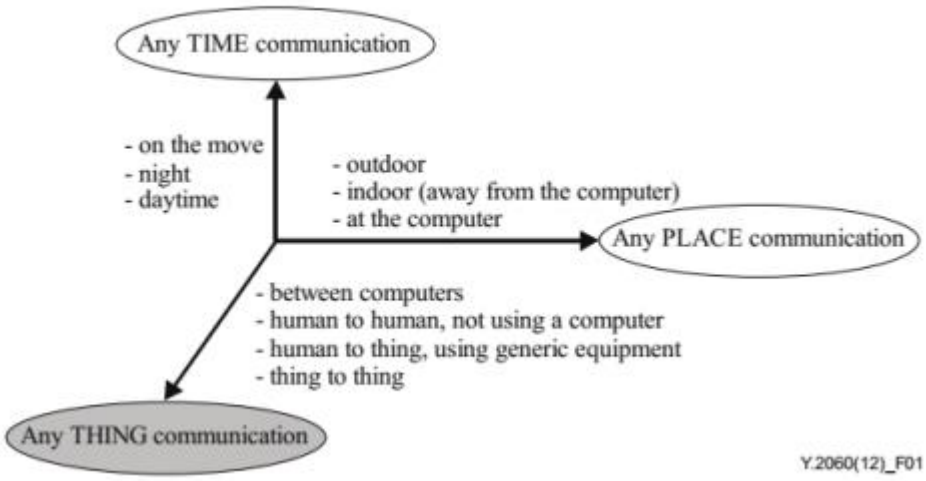

Fig 2.1. “Any thing”, the new dimension added by IoT. [11]

2.1.1. IoT architecture

Unlike in the current iteration of the internet, IoT was designed aiming to provide very diverse objects, both in functionality and technology, with its benefits. To allow such diverse range of devices to communicate with each other, a unified approach to the development of IoT technologies is necessary.

Different standardization organizations and pioneers have suggested different architectural models and approaches, each with an unique approach to the technologies and marketability. This meaning, that there's not a standardized and definitive architecture that every IoT network has to follow.

Even without an standardized architecture,there's a set of components that every IoT project or application will be composed of: Devices with embedded sensors providing the system with information, a network capable of interconnecting said devices, a basic system capable of processing the data and finally an application that translates data into actions or actions into data. These components can be organized into four tiers:

2.1.1.1. Sensor node Tier

The device tier is the physical sensing device that collects data from the real world. Devices can range from small sensors to wearables to large machines, depending on the implementation and the industry. The data itself may be presented in any form.
The device may also display information sent from the application tier.

2.1.1.2. Network and Gateway Tier

In order to allow communication and data transfer between device tier units, a network is required, this network manages the data stream collected by the devices and any other control signal. Depending on the streams, they may come in such diverse forms as mechanical signals or IP-based data streams. On the surface, these streams are completely incompatible. However, when correlating data, a common denominator is needed.

This common denominator is the gateway tier. The gateway tier's manages all the data collected by the system and ensures that it can be made available but outside system's standardized technology sets such as the internet protocol suite.

2.1.1.3. Data Tier

The data tier is where data from gateways is collected and managed and properly stored. Processing is done through analytics, security controls, process modelling and management of devices.

Depending on the type of data and its sensibility, different structures may be called for, such as databases. The management and physical storage of data is a whole classification onto itself simply due to the four volume, variety and availability of the data of a given system.

2.1.1.4. Application Tier

Applications may come in any form, from web servers to mobile applications. In many cases, the application is the user interface that leverages information coming from the analytics tier and presented to the user in a meaningful way and may also offer the user the ability to interact with the devices in runtime. In other cases, the application may be an automation routine that interfaces with other applications as part of a larger function.

2.2. WSN

A WSN is a network of small and inexpensive sensor nodes monitoring physical or environmental conditions and communicating among them using radio signals. This inexpensive sensor nodes can't handle heavy duty communications, hence the need to use low-process consuming standards of communication and data-gathering algorithms.[Fig 2.2.] provides an example of such network and its location within a full-stack solution.

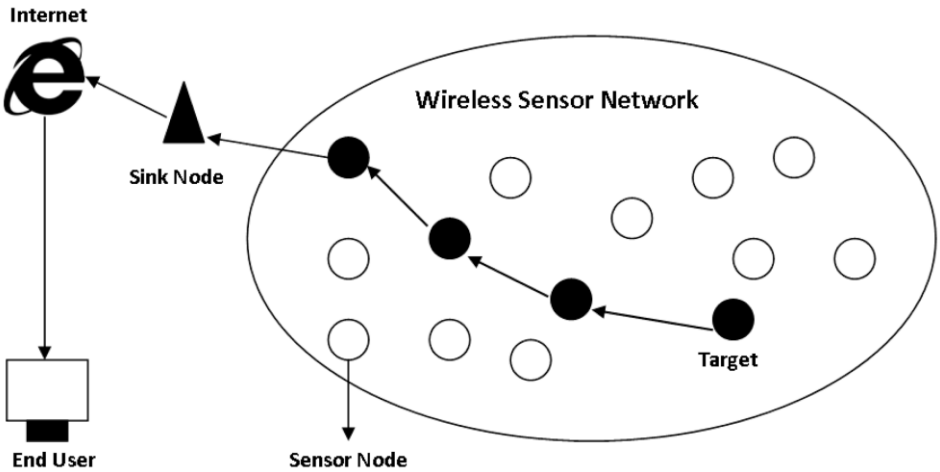

Fig 2.2. WSN topography and routing example. [20]

A wireless sensor node consist of five main components: processor, memory, sensor, radio transceiver and power source. Each component will be described below.

- *Processor*: this unit processes locally sensed information and controls the functionality of other components in the sensor node. The processor generates control messages that direct the sensor to start or stop collecting information about the environment and direct the transceiver to be either in the receiving mode or transmitting mode depending upon the scenario.

- *Memory*: it is used to store both programs (instructions executed by the processor) and data (application related data, raw and processed sensor measurements). Memory requirements are very much application dependent.

- *Sensor*: this component perform the functions of input devices collecting information of ambient conditions and converting it into electrical signal. This signal is often digitized by an Analog to Digital Converter (ADC) and then sent to the processor for further data management.

- *Radio*: This component is used to exchange data between devices and integrates the functionality of both transmitter and receiver combined into a single device known as transceivers. WSN nodes include a low-rate, short-range wireless radio. Typical rates are 10-100 kbps, and range is less than 100 meters.

- *Power source*: provides energy to make functional the node. Rechargeable batteries are the predominant mean for providing energy to sensor nodes today and will be used in this project. Current sensor nodes may be equipped with energy harvesting mechanisms, which are able to renew their energy from solar, thermogenerator, or vibration energy. This methods increase sensor nodes autonomy.

2.3. IPv6

IPv6 is the next generation Internet Protocol (IP) standard intended to eventually replace IPv4, the protocol many Internet services still use today. Every computer, mobile phone, and any device connected to the Internet needs a numerical IP address in order to communicate with other devices. The original IP address scheme, called IPv4, is running out of addresses, thus the need for a new protocol, IPv6. This is due to the fact that IPv6 addresses are based on 128 bits whereas IPv4 addresses are based on 32 bits.

Apart from increasing the pool of possible addresses, there are other important technological changes in IPv6 that will improve the whole functionality of the IP protocol:

- No more NAT (Network Address Translation)

- Auto-configuration of addresses.
- Unique private addresses, so no private address collisions.
- Better multicast routing.
- Simpler header format.
- Simplified, more efficient routing.
- True quality of service (QoS), also called "flow labeling".
- Built-in authentication and privacy support.
- Flexible options and extensions.
- Easier administration (say good-bye to DHCP).

IPv6 is set to be the next step of the internet protocol and it's being adopted at a quick pace worldwide, reaching around 23% adoption as of September 2018 as seen in [Fig 2.3.]

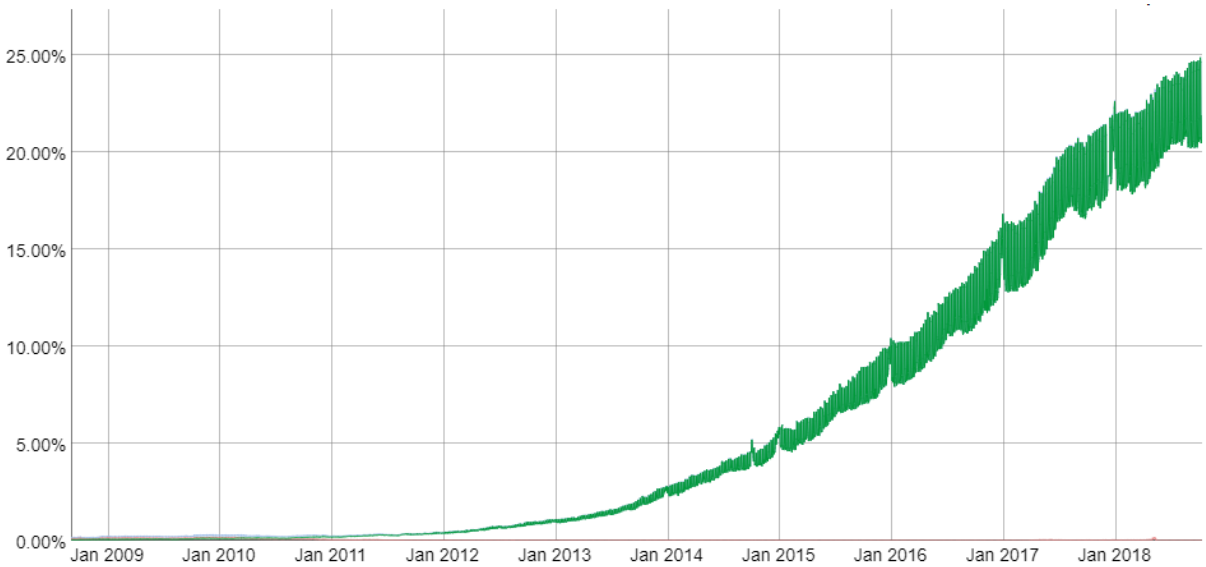

Fig 2.3. Adoption of IPv6 Worldwide. [5]

2.4. IEEE 802.15.4

IEEE 802.15.4 is a technical standard, specified in RFC-4944, which defines the operation of low-rate wireless personal area networks (LR-WPANs). It specifies the physical layer and media access control for LR-WPANs, and is maintained by the IEEE 802.15 working group, which defined the standard in 2003. It can be used by 6LoWPAN, the technology used to deliver the IPv6 version of the Internet Protocol (IP) over WPANs, to define upper layers like Thread.

The IEEE 802.15.4 standard is one of the most used in WSN. As a consequence of its low power consumption, it also presents low data rates and transmission power. Therefore, this wireless technology only allows short-range

communications, typically with distances up to 100 meters. Several other standards or stack specifications use IEEE 802.15.4 as their physical layer (PHY) and data link-layer (DLL), including 6LoWPAN, ISA100 and ZigBee specifications. [Fig 2.4.] compares the IP protocol stack to a variety of possible 6LoWPAN stacks.

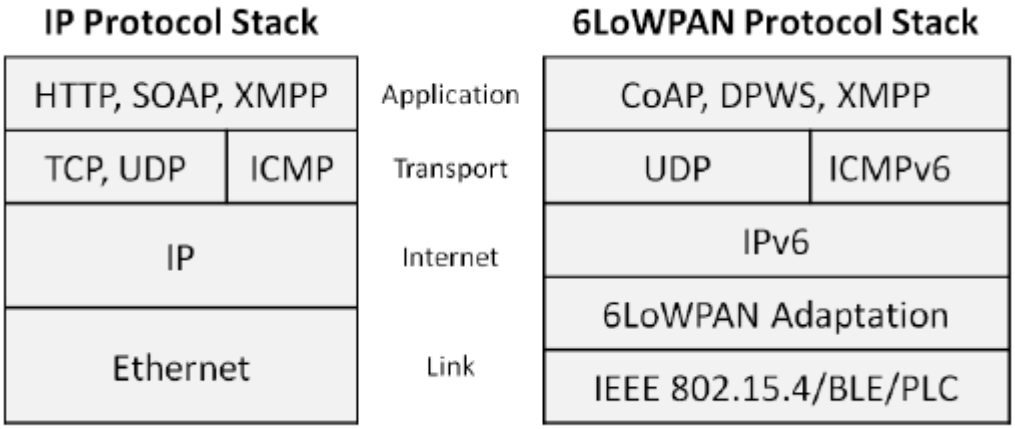

Fig 2.4. Role of the IEEE 802.15.4 standard in a 6LoWPAN network. [8]

The IEEE 802.15.4 standard is designed to operate in the 2.4 GHz band at a rate of 250 kbps, at a rate of 40 kbps in 868 MHz band and at a rate of 20 kbps in 915 MHz band. Despite having these three alternatives, the 2.4 GHz band is usually used, as it is common throughout the world, while the other two frequencies are only available in North America (915 MHz) and European Union (868 MHz) respectively.

In a 802.15.4 architecture, devices are conceived to interact with each other in a conceptually simple wireless network. It's network layers are based on the OSI model even though only the lower layer are defined in the official RFC, leaving up to the developer which higher level protocol may suit the project the best.

2.5. 6LoWPAN

6LoWPAN (IPv6 over low-power personal area wireless networks), is a unique protocol designed to allow the transmission of IP packets in networks based on the IEEE 802.15.4 standard. It is also designed for applications with a very low data flow and very limited resources, such as WSNs and most IoT applications.

The concept was created because engineers felt like the smallest devices were being left out from the Internet of Things. 6LoWPAN can communicate with

802.15.4 devices as well as other types of devices on an IP network link like WiFi. A bridge device, acting as 6LBR(6LoWPAN border router), can connect the two. [Fig 2.5.] explains where 6LoWPAN technologies are used within an heterogeneous network.

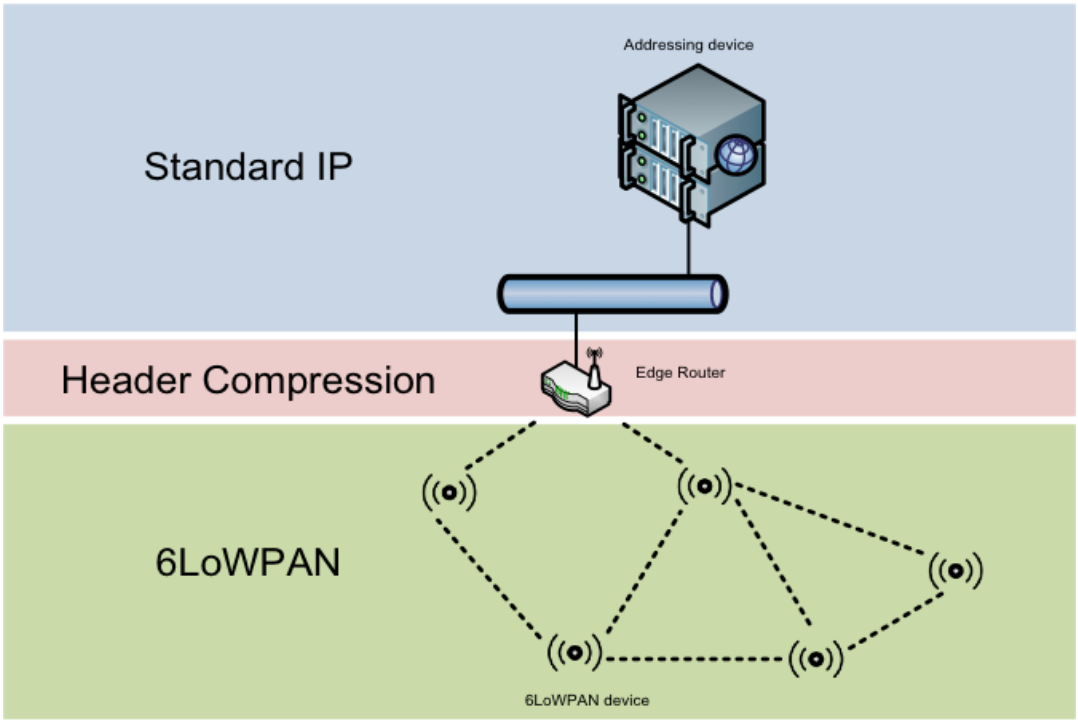

Fig 2.5. 6LoWPAN performance setting. [10]

2.5.1. RPL

RPL is a protocol that consists of routing techniques that are organized in Directed Acyclic Graphs (DAG) messages. DAG is a structure in which all nodes are connected, but there is not a single round-trip path available from one node to another. The nodes are connected to each other and all messages are routed to a node forming new structures called DODAG (destination oriented DAG). The node that collects all DODAG messages is called root or sink like node a in [Fig 2.6.]

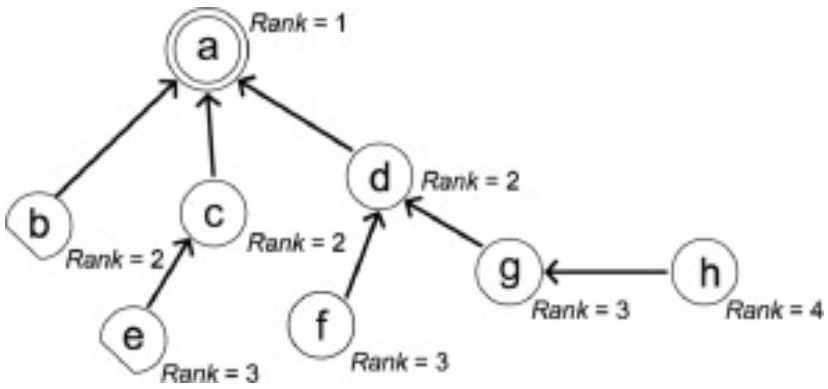

Fig 2.6. Example of a DODAG Topology. [25]

The formation of the DODAG structure starts at the root. The root transmits a DODAG Information Object (DODAG) message to its neighbors. Each node in the network has its own rank representing its level in the network. The range is a scalar representation of the position of the nodes in the current DODAG and does not allow loops in the network. After receiving the DIO message from the root, each node calculates its own range and sends its DIO messages with the new rank values to its neighbors so they can calculate their own ranges. The range increases through the network with increasing distance to the root. The root DODAG has the option to start the nodes rank calculation again. To initiate recalculation, the root sends DIO messages with the sequence number greater than the previous message and advertise the other nodes that it wants to redo the DODAG structure.

When a node joins the DODAG structure the first time, it will have to wait listening a DIO message from the nearest node. A node can also transmit its own DODAG Information Solicitation (DIS) message from the nearest node.

One of the fields in the DIO message has information about the objective function. The objective function is used to calculate the cost of sending messages by the link to the root and to ensure that the message is sent through the link with the lowest cost. There are many types of objective functions and all nodes in the network have to know the type of objective function used. Depending on the objective function and the ranks of its neighbors received in the DIO message, the nodes could either join the DODAG or remain in the existing DODAG. The objective function depends only on the current application being used in the network, however does not reduce the possibilities of using RPL.

RPL traffic supports bidirectional messages, making it possible to send messages from the nodes to the sink and from the sink to the nodes. Messages that send nodes to the sink with upward path through DODAG, choose their path by selecting "preferred parent." The DODAG root in DIO messages may have advertised a set of destination prefixes to which it provides connectivity. These prefixes can be used to populate network routing tables or as a default route through preferred parents. RPL is designed as a multipoint-to-point (MP2P) communications protocol for delivering data from the network nodes to the DODAG root (or sink in WSNs). With additional traffic, RPL can also use point-to-multipoint (P2MP) and point-to-point (P2P) communications. The additional traffic means that RPL would have to have some other type of messages, in addition to DIO messages.

DIO messages are used to configure and maintain the network and the DIS message is used to add new nodes to the DODAG. Another type of message that RPL defines is the DAO (Destination Advertisement Object). DAO messages are

sent from a node to the root and allow the nodes to announce their presence to the nodes that are between the node and the root. Each node transmits the DAO message to its parent, which writes its own address in the received message and forwards it to its own parent. The same procedure is repeated in all the path to the root. The parent receiving the first DAO message from a node is called the preferred parent and for each node there can be no more than one preferred parent. If the DODAG root receives the DAO message, it reads the addresses of the nodes in reverse order, thus creating the path from the node to the root. The root saves this path in its memory and uses it whenever it wants to send a message to that node.

In short, RPL is a protocol that provides rapid network configuration and knowledge distribution across the network and effective network adaptation even if the network topology changes. It is recommended for use in smart grids and WSNs scenarios.

2.5.2. 6LoWPAN Main Functions

As with all link-layer mappings of IP, RFC4944 provides a number of functions. Beyond the usual differences between L2 and L3 networks, mapping from the IPv6 network to the IEEE 802.15.4 network poses additional design challenges (see RFC 4919 for an overview).

- **Adapting the packet sizes of the two networks.** IPv6 requires the maximum transmission unit (MTU) to be at least 1280 bytes. In contrast, IEEE 802.15.4's standard packet size is 127 bytes. A maximum frame overhead of 25 bytes spares 102 bytes at the media access control layer.

- **Address resolution.** IPv6 nodes are assigned 128 bit IP addresses. IEEE 802.15.4 devices may use either of IEEE 64 bit extended addresses or, after an association event, 16 bit addresses that are unique within a PAN. There is also a PAN-ID for a group of physically collocated IEEE 802.15.4 devices.

- **Differing device designs.** IEEE 802.15.4 devices are intentionally constrained in form factor to reduce costs, reduce power consumption and allow flexibility of installation. On the other hand, wired nodes in the IP domain are not constrained in this way; they can be larger and make use of mains power supplies. This enables 6LoWPAN networks to be mobile and low-power while being relatively cheap.

- **Differing focus on parameter optimization.** IPv6 nodes are geared towards attaining high speeds. Algorithms and protocols implemented at the

higher layers such as TCP kernel of the TCP/IP are optimized to handle typical network problems such as congestion. In IEEE 802.15.4-compliant devices, energy conservation and code-size optimization are the main focus of its algorithms.

- **Adaptation layer for interoperability and packet formats.** An adaptation mechanism to allow interoperability between IPv6 domain and the IEEE 802.15.4 can best be viewed as a layer problem. Identifying the functionality of this layer and defining newer packet formats, if needed, is an enticing research area. RFC 4944 proposes, but doesn't define, an adaptation layer to allow the transmission of IPv6 datagrams over IEEE 802.15.4 networks.

- **Device and service discovery**. Since IP-enabled devices may require the formation of ad hoc networks, the current state of neighboring devices and the services hosted by such devices will need to be known. IPv6 neighbour discovery extensions is an internet draft proposed as a contribution in this area.

- **Security.** IEEE 802.15.4 nodes can operate in either secure mode or non-secure mode. Two security modes are defined in the specification in order to achieve different security objectives: ACL (Access Control List) and Secure mode. Secure mode provides confidentiality of the frame along with the message integrity, access control, and sequential freshness.

2.6. Contiki

Contiki is an open source operating system for the Internet of Things. Contiki connects tiny low-cost, low-power microcontrollers to the Internet. Contiki is a powerful toolbox for building complex wireless systems. It was developed at the Swedish Institute of Computer Sciences by Adam Dunkels et al. It is written in the C programing language and all programs for it are also in C. Contiki is a highly portable OS and it has already been ported to several platforms running on different types of processors. The most platforms often use Texas Instruments MSP-430 as well as Atmel ATmega series of microcontrollers.

Contiki uses event-driven programming model to handle concurrency. All processes share one stack, allowing savings of memory. It is the main advantage of the Contiki. Protothreads are used to realize this model. Protothreads provide conditional and unconditional blocking wait and they use local continuations to save the state when it blocks. When the Protothread is resumed, it jumps back to the next instruction.

Contiki supports both IPv6 and IPv4 stack implementations, along with the recent low-power wireless standards: 6lowpan, RPL, CoAP. It also uses Rime stack. It is a lightweight communication stack for sensor networks and it has thinner layers than traditional stacks. The layers are simple and they have small headers (only a few bytes). Rime also supports code reusing and the main purpose of this protocol is to simplify the implementation of sensor networks.

2.6.1. Instant Contiki

Instant Contiki is an entire Contiki development environment in a single download. It is an Ubuntu Linux virtual machine that runs in VMWare player and has Contiki and all the development tools, compilers, and simulators used in Contiki development installed.

In the making of this project, an Instant Contiki 2.7. Distribution was used, due to its ease of installation and the convenience of its availability. The process of installation will be explain and further discussed in section [3.1].

2.7. Hardware used

In this chapter the physical devices used in the project will be introduces and explain in detail in order to fully understand its limitations and the implications that may come with them when implementing the network design in a real-world testing scenario. More information about this devices is provided in [Table 2.1.]

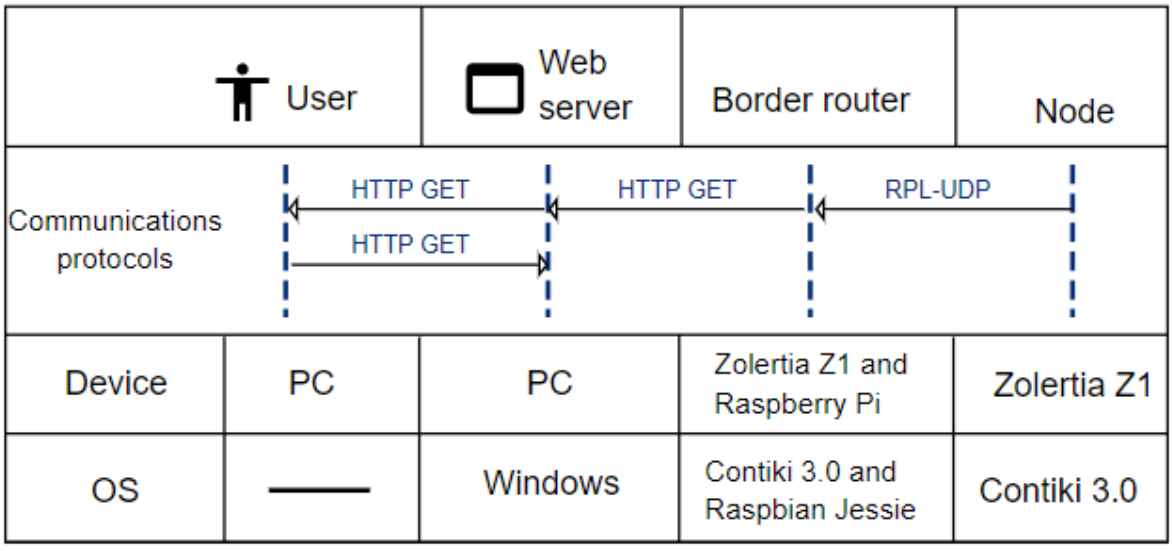

Table 2.1. Devices information

2.7.1. Zolertia Z1

The Z1 is a low power wireless module compliant with IEEE 802.15.4 and Zigbee protocols intended to help WSN developers to test and deploy their own applications and prototypes with the best tradeoff between time of development and hardware flexibility. Its core architecture is based upon the MSP430+CC2420 family of microcontrollers and radio transceivers by Texas Instruments, which makes it compatible with motes based on this same architecture. However, the MCU present in Z1 is the MSP430F2xxx instead of the MSP430F1xxx, as is customary among other motes, like Crossbow's TelosB, Moteiv's Tmote, and alike. This fact entails subtle differences due to inner changes between F1xxx and F2xxx devices, but these differences are not expected to arise at the firmware application level if a supported operating system is employed when developing. Some of Zolertia's features can be seen in [Fig 2.7.] and will be further explained in chapter [2.7.1.1.]

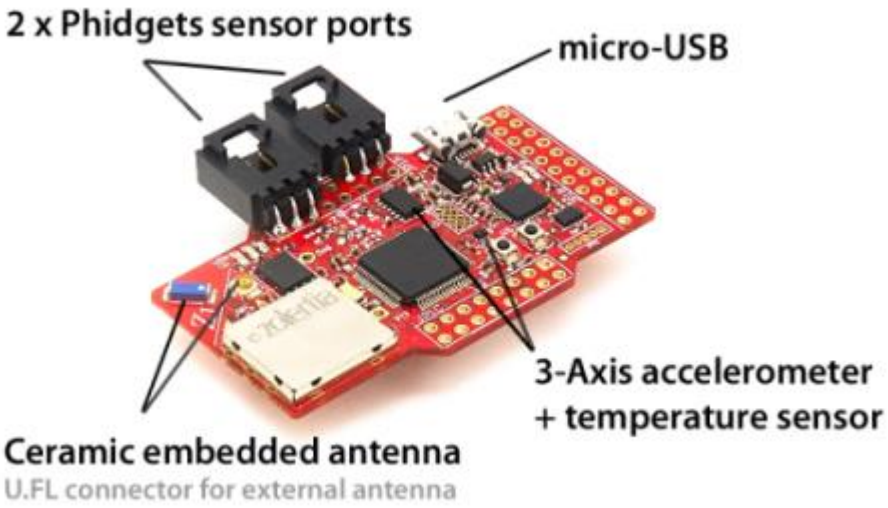

Fig 2.7. Zolertia Z1 and some of its features. [16]

2.7.1.1. Zolertia Z1 features

- Ultra-Low Power MCU and 2.4GHz Transceiver.
- Totally compatible with Contiki OS.
- 2 x Digital Built-in sensors (temperature and 3-axis accelerometer).
- USB Programming Ready.
- Flexible Powering: Battery Pack (2xAA or 2xAAA), Coin Cell (up to 3.6V), USB Powered, Directly Connected through two wires coming from a power source. USB VCC and GND pins are available on the digital buses expansion port. You can connect to this pins any power of source from 4V to 5.25V and it will be regulated to 5V and 3V.

2.7.2. Raspberry Pi

Raspberry Pi (RPI) is a low cost single board computer (SBC) developed in the United Kingdom by the Raspberry Pi Foundation. As well as a computer, it has input and output peripherals and its official operating system is an adapted version of Debian, called Raspbian that will be further explained in section [2.7.2.2]. Raspberry Pi is widely used in development of IoT applications, due its low cost and reduced size. There are some Raspberry Pi commercial models, in this project the model used is Raspberry Pi B with Raspbian installed. [Fig 2.8.] shows a picture of Raspberry Pi B.

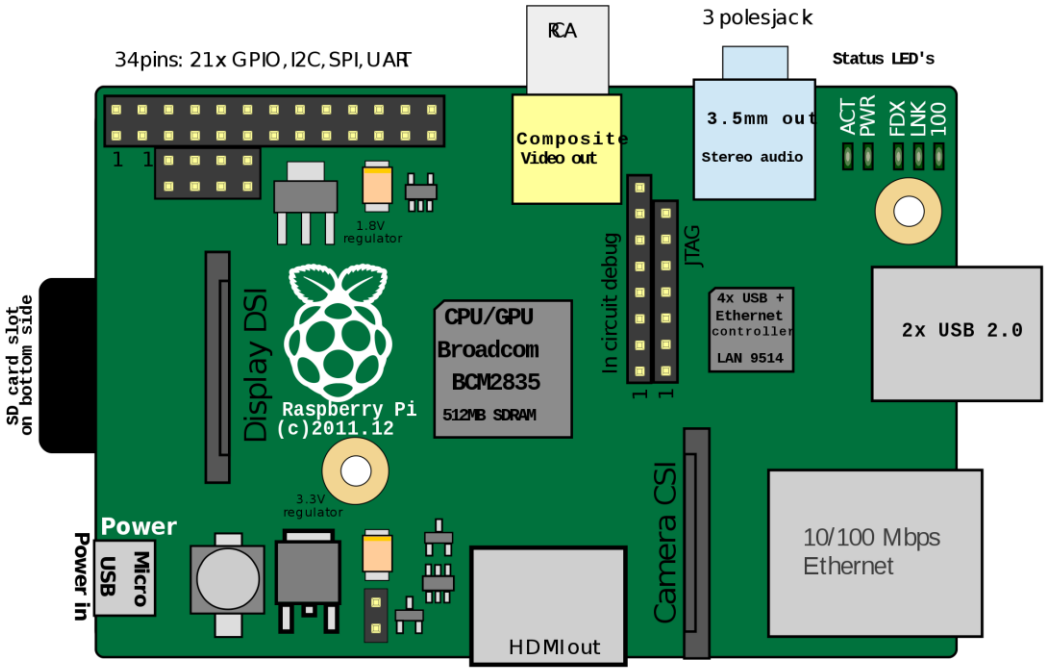

Fig 2.8. Raspberry Pi B and some of its features. [17]

2.7.2.1. Raspberry Pi-B technical specifications

The main technical specifications of the Raspberry Pi model to be used in this project are:

- Broadcom BCM2835 SoC
- 700 MHz ARM1176JZF-S core CPU
- Broadcom VideoCore IV GPU
- 512 MB RAM
- 2 x USB2.0 Ports
- Video Out via Composite (PAL and NTSC), HDMI or Raw LCD (DSI)
- Audio Out via 3.5mm Jack or Audio over HDMI
- Storage: SD/MMC/SDIO
- 10/100 Ethernet (RJ45)

- Low-Level Peripherals:
  - 8 x GPIO
  - UART
  - I2C bus
  - SPI bus with two chip selects
  - +3.3V
  - +5V
  - Ground
- Power Requirements: 5V @ 700 mA via MicroUSB or GPIO Header
- Supports Debian.

2.7.2.2. Raspbian

Raspbian is a free operating system based on Debian optimized for the Raspberry Pi hardware. An operating system is the set of basic programs and utilities that make your Raspberry Pi run. However, Raspbian provides more than a pure OS: it comes with over 35,000 packages, pre-compiled software bundled in a nice format for easy installation on your Raspberry Pi.

The initial build of over 35,000 Raspbian packages was completed in June of 2012. However, Raspbian is still under active development with an emphasis on improving the stability and performance of as many Debian packages as possible.

Raspbian Jessie, specifically, is the OS that will be running on Raspberry Pi for this project. A well-thought and in-detail guide on how to properly install Raspbian and how to configure a proper working environment will be provided in chapter [3.2]

3. DESIGN AND IMPLEMENTATION

3.1. Instant Contiki installation

Instant Contiki is the OS chosen to run on the Zolertia Z1 motes used in this project. This is a step-by-step guide guide to get a working virtual machine with a distribution of Instant Contiki running and properly configured running in your computer.

- 1. Download Instant Contiki.
- 2. When downloaded, unzip the file, place the unzipped directory on the desktop.
- 3. Download and install VMWare Player. It is free to download, but requires a registration. It might require a reboot of your computer.
- 4. Start Instant Contiki by running InstantContiki3.0.vmx.
- 5. Wait for the virtualUbuntu Linux boot up. Log into Instant Contiki. The password is user.

3.1.1. Compiling an application on Contiki.

To compile an application in Contiki there are 3 files: the Contiki application file, explained in the previous section, which contains the application to be used, the optional configuration file of our application and the Makefile.
The Makefile structure used by Contiki is quite simple:

```
CONTIKI = ../../../..
all: app-name
include $(CONTIKI)/Makefile.include
```

The first line indicates the location of the Contiki source root, the second linespecifies the applications to be compiled, and the third line includes the entireContiki system, which contains the Contiki core definitions and points to the specific Makefile to our target platform. A project in Contiki may have an optional configuration file called projectconf.h. This file is not active by default, to activate it, the following line must be added to the Makefile file of the project:

```
CFLAGS += -DPROJECT_CONF_H=\"../project-conf.h\"
```

The project-conf.h file can activate a number of Contiki configuration options or modify default Contiki parameters.

To compile an application, target hardware platform must be specified using the TARGET = syntax platform, and if it is not specified by default the native platform will be compiled.

```
make TARGET= z1
```

To compile and load the application on Z1 node:

```
make TARGET=z1 BOARD=remote-revb foo.upload
```

Further information on the commands that were required and used in the making of this project will be given in the chapters to come.

3.2. Raspbian installation

Raspbian is the chosen OS to run the Raspberry Pi, its main function will be to enable communications between IP networks and the WSN composed, in this case, of 4 Zolertia Z1 motes. In this chapter a detailed step-by-step guide on how to get a Raspbian copy running on our Raspberry Pi will be provided.

Before beginning, note that an SD card or micro SD card with and SD-card adapter, an spare mouse, a keyboard and an HDMI connection screen will be required.

1. Download Raspbian from the original web page [22]
2. Download an SD-burning program such as Ember.
3. Using the previously downloaded program burn the Raspbian OS image into the SD card.
4. Connect the previously stated devices and the SD card into the Raspberry Pi and then plug it to DC current, a mobile charger able to provide 1 A should do the trick.
5. A configuration wizard should appear on your screen, wait for it to install every package needed and done.

3.3. Functional design

In this chapter, the global design of the final solution will be discussed in an attempt to give a global picture of the project. Then, in the sections to come, a more in-detail explanation of the design decisions and its implementation will be given. In order to get a global idea of the proposed solution, [Fig 3.1.] provides information about how the devices interact and which communication interfaces and protocols were used

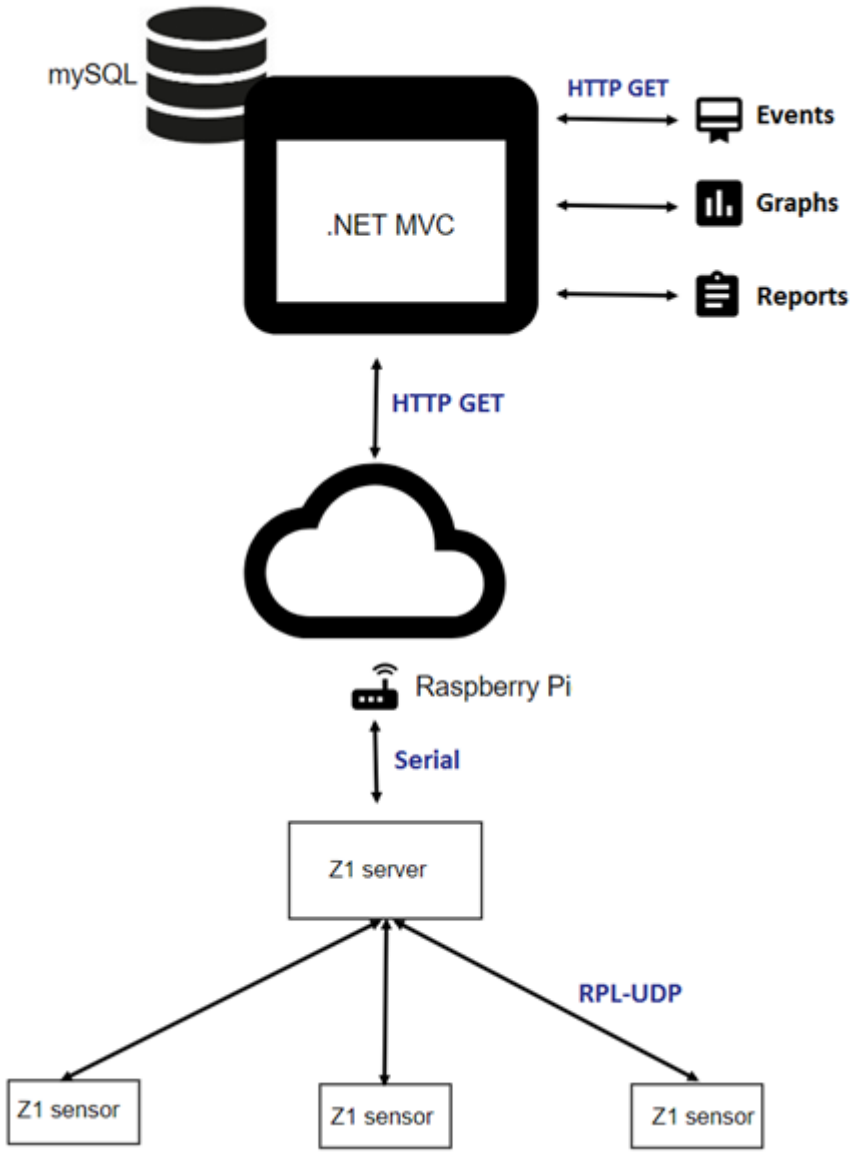

Fig 3.1. Schematic of the whole solution including devices and comunications.

As mentioned in chapter [2.7.1], Z1 nodes have built in sensors that can be used to track physical world signals. In this project, temperature, battery and acceleration were tracked.

Each node reads the sensor values and after getting enough data to send a packet, the data is transmitted to the Border Router through UDP.

In order to allow the access to this data from outside IP networks, the border router is going to run a series of scripts, that will allow the data sensed by the WSN nodes to be sent to a ASP.NET server located in the cloud and accessible to everyone.

This ASP.NET will implement a direct path to a mySQL database as means of persistency if further analysis is considered adequate. This web server will also have an user interface allowing users to read data in real-time and history of relevant events captured by the WSN.

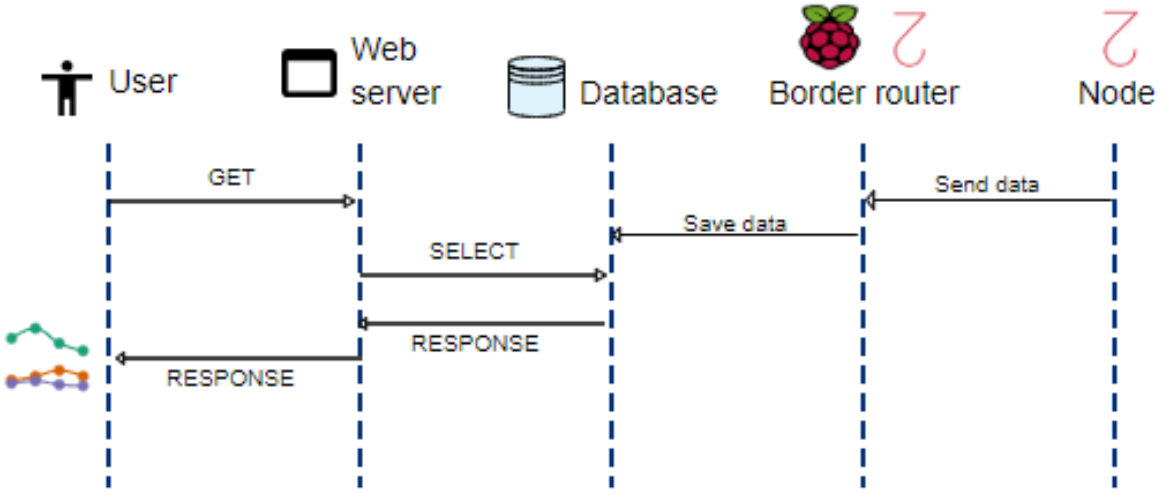

Fig 3.2. Communications flow within the system

As seen in [Fig 3.2.] the data sensed by the WSN is added to the database independently of the web server, this increases the modularity of the solution and allows multiple web servers to feed from this same database if desired. It is also worth noting that the Border router interface is composed by both a zolertia Z1 and a Raspberry Pi operating as a single border routing unit.

It is also worth noting that the bordering router connecting the WSN and IP realms is composed of two devices, a raspberry Pi and a Zolertia Z1, more context on that will be provided on chapter [3.5.].

3.4. Node design

Starting from the bottom of the diagram and at the root of the projects architecture, there's the WSN node. The nodes are running Contiki, explained in detail in chapter [2.6]. Z1 nodes have two built-in digital sensors: temperature and 3-axis accelerometer, and also a built-in voltage sensor provided as an ADC (Analog to Digital Converter) input channel. For this project, all of the previously mentioned sensors were used.

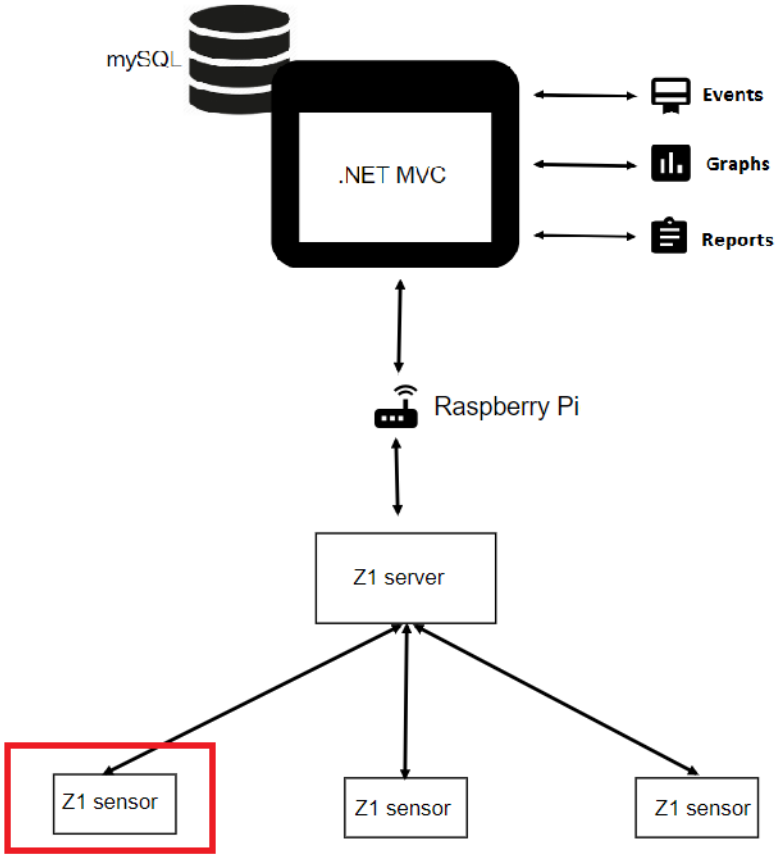

Fig 3.3. Location of a node inside the whole system

Each node read the sensors values and send them to the border router through RPL-UDP once enough data is captured and the size of every single data triplet optimized.

According to 802.15.4 standard, the maximum size of a payload in this kind of network is 102 bytes. Taking that into account, and the fact that some of the data sensed could vary in size, the maximum amount of samples per UDP packet sent was calculated as follows:

Maximum Payload size = 102 Bytes

ID size = 1 Byte
Temperature data size = 2 Byte
Acceleration data size = 3 or 4 Bytes
Battery Data size = 3 or 4 Bytes
Needed separators = 4 Bytes

Total Data Size= 13-15 Bytes

Max data triplets per UDP packet = floor($\frac{102\ Bytes}{15\ Bytes}$) = **6**

After simple calculations, the final decision was to send 6 data triplets for each UDP packet sent from the sensor node to the border router. The data structure can be seen in [Fig 3.4.]

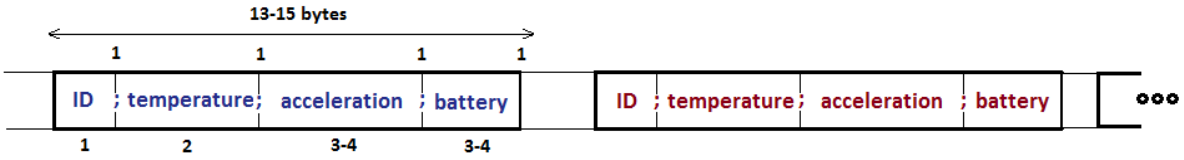

Fig 3.4. Data triplet size and structure

This means that the program is going to measure battery, accelerationa and temperature 6 times for every UDP transmission.

3.4.2. Program architecture

A C program was developed with the only purpose to meet this network requirements.

The function get_and_save_data() is called on every duty cycle. It's a multi purpose function which reads the values of the three sensors previously mentioned, processes the data on a superficial level, and then stores it on a buffer, where the data is gonna wait to be sent when the given conditions stated in the main thread are met.

It is also worth noting that every one of each sensor node is running an slightly modified version of the same code, in which the package is marked with an Id, this Id is later used to identify to which one of the sensor node devices is transmitting the data.

```c
static void get_and_save_data(void)
{
  uint32_t aux;

  if(payload != NULL) {
      //Measure battery
    bateria = battery_sensor.value(0);
    mv = (bateria * 2.50 * 2) / 4096;

    //Measure Temperature
    raw = tmp102_read_temp_raw();
    absraw = raw;
    if (raw < 0) {
      absraw = (raw ^ 0xFFFF) + 1;
    }
    tempint  = (absraw >> 8) ;

    //Measure Acceleration
    z_acc = accm_read_axis(Z_AXIS);

    //Save Data into Buffer
    sprintf(temp, "b;%d;%i;%d|",  tempint, bateria, z_acc);
    strcat(payload, temp);
    counter++;   //Add 1 to number of sample packets in  buffer
  }
  else {printf("No memory available");}
}
```

Code snippet 3.1. get_and_save_data() function

The function send_packet() is not called on every duty cycle, only when the conditions stated in the main thread are met. It's purpose is to send the data stored in the buffer directly to the border router through UDP and then reset the buffer and the data triplet counter.

```c
static voidsend_packet(void)
{
  static int seq_id;
  char buf[MAX_PAYLOAD_LEN] ;
  //Copy buffer to a temporal variable to send
  strcpy(buf,payload);
  //Restart packet counter
  counter = 0;
  //Restart buffer
  strcpy(payload,"");
  //In case you want to track packet numbers
  seq_id++;
  //Send Data
  uip_udp_packet_sendto(client_conn, buf, strlen(buf),
                        &server_ipaddr, UIP_HTONS(UDP_SERVER_PORT));
}
```

Code snippet 3.2. send_packet() function

The main thread is the entry point of our code and determines which functions get called. In the nodes main thread, the udp connections get created and bound, the sensors are initialized and then there's an infinite loop where the get_and_save_data() function is gonna get called on every iteration and the function send_packet() is gonna get executed every SAMPLES_PER_PACKAGE=6 iterations of the loop.

```
PROCESS_THREAD(udp_client_mod_process, ev, data)
{
  PROCESS_BEGIN();
  PROCESS_PAUSE();

  set_global_address();
  tmp102_init();
  SENSORS_ACTIVATE(battery_sensor);
  accm_init();
  //Create Connection
  client_conn = udp_new(NULL, UIP_HTONS(UDP_SERVER_PORT), NULL);
  //Bind connection
  udp_bind(client_conn, UIP_HTONS(UDP_CLIENT_PORT));

  etimer_set(&periodic, SEND_INTERVAL);

  while(1) { //Infinite loop
    PROCESS_YIELD();
    if(ev == tcpip_event) {
      tcpip_handler();
    }
    //Measure and save data
    get_and_save_data();

    if(etimer_expired(&periodic)) {
      etimer_reset(&periodic);

        if (counter == SAMPLES_PER_PACKAGE){ //Send Packet if buffer is full
        ctimer_set(&backoff_timer, SEND_TIME, send_packet, NULL);
        }
     }
  }
  PROCESS_END();
}
```

Code snippet 3.3. Main thread

The behaviour of this main thread can be easily comprehended when looking at a flowchart [Fig 3.5.]

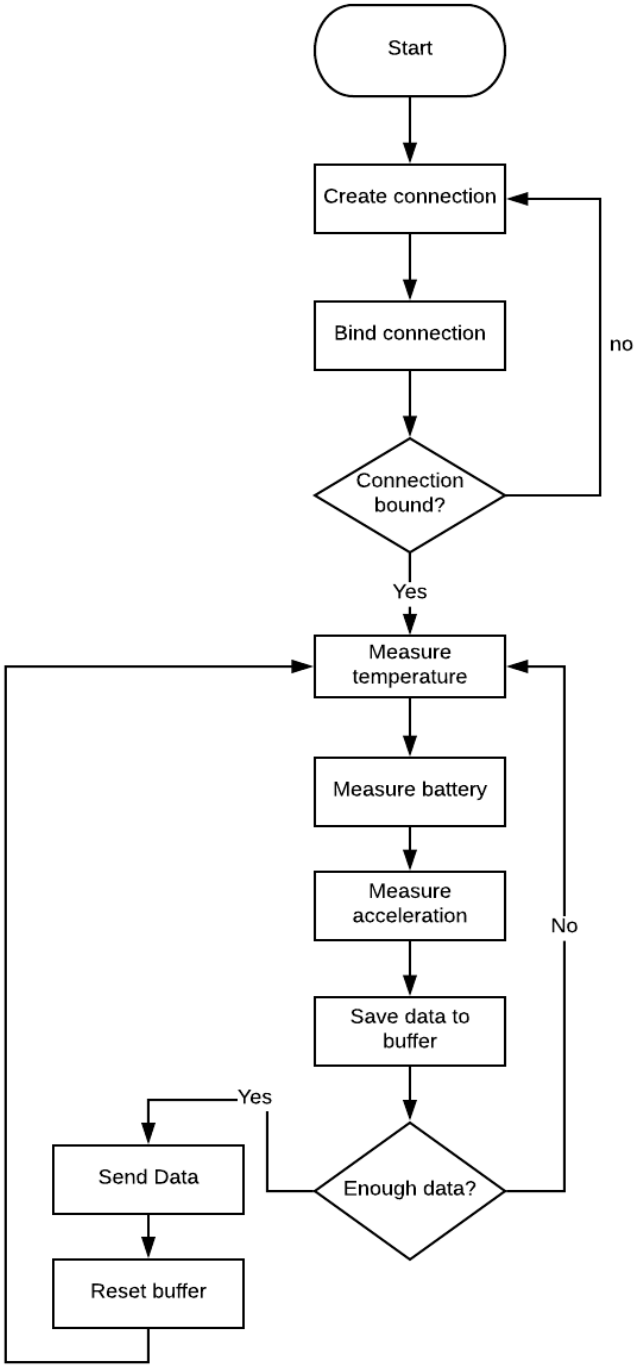

Fig 3.5. Node program flow chart

3.5. Border router design

In this projects design, the border router is composed of two physical objects, a Zolertia Z1 device acting as an UDP Server and a Raspberry Pi.

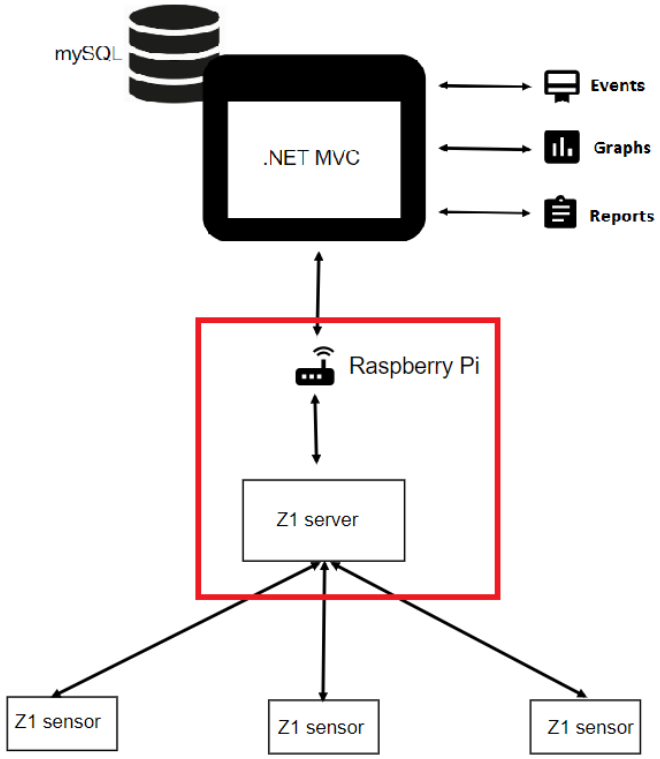

Fig 3.6. Location of the border router inside the whole system

The first of the devices is going to act as an UDP server and receive all the data that Z1 sensor nodes send, then, the data is going to be transmitted to the Raspberry Pi via serial.

As for the Raspberry Pi, before beginning the development phase, an open source Operating system, Raspbian Jessie, was properly installed. More information on how this process was done can be found in chapter [3.2.].

After installing the OS and configuring the development environment, a python script was developed and configurations were set to allow for this script to run as the Raspberry Pi device turns ON.

To do so, a simple launching script launchscript.sh was generated:

```
#!/bin/sh
sleep 5
sudo python border-router-script.py
```

Code snippet 3.4. Python script launcher

In order for this newly launcher script to be run on init a new line was added to the /etc/rc.local file:

```
/Desktop/bin/launchscript.sh &
```

A python script has been developed for the Raspberry Pi and executes as the device turns on. Incoming data from the Z1 server is going to be read by the Raspberry Pi serial port and processed by said python script, after superficial processing and securing data integrity, the information from the nodes is going to be forwarded to the web server using the HTTP GET method.

3.5.2. Program architecture

In order to implement the border router design previously explained, two programs will be developed, one C program that will be running on the Zolertia Z1 device acting as an UDP server and a Python program running on the Raspberry Pi.

The main function of the Z1 program is called tcpip_handler(). It reads incoming messages and in case a new UDP packet is received, the data is forwarded using the serial port. An end of line command is then issued to allow for an easier parsing by the Raspberry Pi.

```
1  static void
2  tcpip_handler(void)
3  {
4    char *appdata;
5    //If new data received
6    if(uip_newdata()) {
7      appdata = (char *)uip_appdata;
8      appdata[uip_datalen()] = 0;
9      //Send data through serial.
10     PRINTF("%s", appdata);
11     PRINTF("\n");
12   }
13 }
```

Code snippet 3.5. tcpip_handler() function

The code running in the Raspberry Pi is somewhat more complex. The program will start when the Raspberry turns on and will listen to the default serial interface, if connection to the Z1 device is established, the program will start reading incoming packets coming from the serial as bytes.

After making sure that this data is relevant, the byte array will be parsed as an as a string and sent to the web server via an exposed REST endpoint.

If any problem is found during the execution of this code, an attempt to Log the data to the database will be made.

```
import serial, string, requests, time
counter = 0
total_strings = ""
output =" "

#Packet blocks to send to the server
blocks_per_packet=25

# IP Address of the web server
ip_webserver ="192.168.1.110"

ser = serial.Serial('/dev/ttyUSB0',115200, 8, 'N', 1, timeout=1)
if ser != null:
    while True:
        print ("----")
        while output != "":
            output =ser.readline().decode("utf-8")
            if len(output) > 5 :
                counter=counter+1
                total_strings+=output
                if (counter>=blocks_per_packet):
                    counter = 0
                    r = requests.get(("http://192.168.1.110/data/DataFromRPI?data="+total_strings).replace("\n","").replace("\r",""))
                    total_strings=""
            output = " "
else print("No serial device Found")
r = requests.get("http://192.168.1.110/data/Logger?data=RPI Border router error: No serial device found");
```

Code snippet 3.6. Raspberry Pi python function

This code can be easily understood by taking a glance at [Fig 3.7.], where a flow chart is provided.

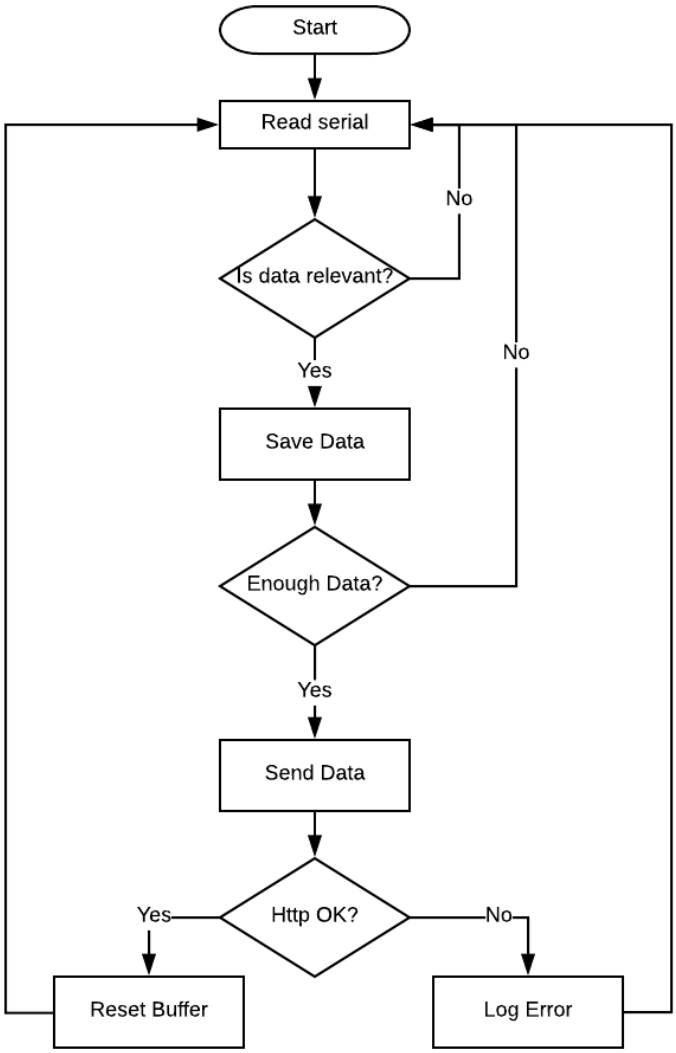

Fig 3.7. Border router program flow chart

3.6. Database

The large amount data collected by the Zolertia Z1 devices needs to be stored in some kind of non-dynamic way due to its raw size and order to do so a database was used. The technology employed for this purpose is mySQL. MySQL is an Oracle-backed open source relational database management system (RDBMS) based on Structured Query Language (SQL). MySQL runs on virtually all platforms, including Linux, UNIX and Windows.

This database was initially meant to be running on the Raspberry Pi, but due to its lack of resources and the demanding needs of the high-speed WSN it was finally implemented in a personal computer along with the web server.

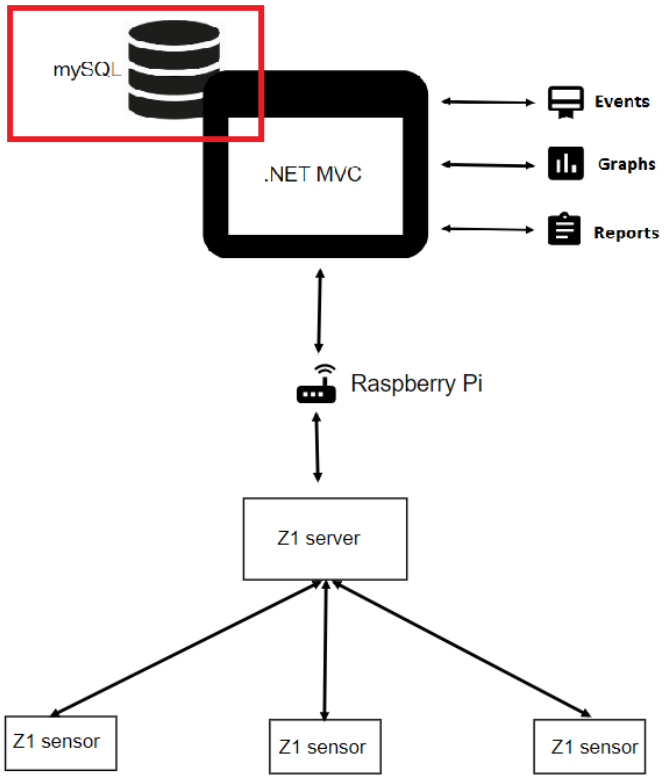

Fig 3.8. Location of the database inside the whole system

Database structure is simple, it has two tables: one table for storing temperature and battery values along with the date and time, and a second table to store information about any error that might have happened along the way from the node to any of the user interfaces. This tables can be seen in [Fig 3.9.]

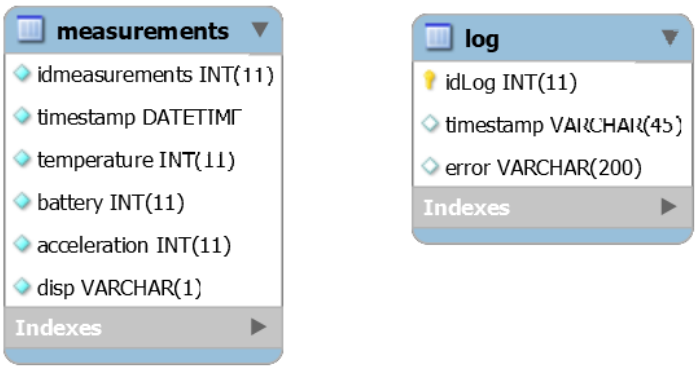

Fig 3.9. Database tables and its properties

The flow that allows data from the raspberry Pi to be persisted into the database is also pretty simple, as long as the message from the Raspberry Pi is properly parsed, the data will be inserted into the *measurements* database. If for any reason the data is corrupted or an unexpected string has caused an error, said

error will be logged and inserted into the *log* along with the Exception message or a detailed message of where the error occured.

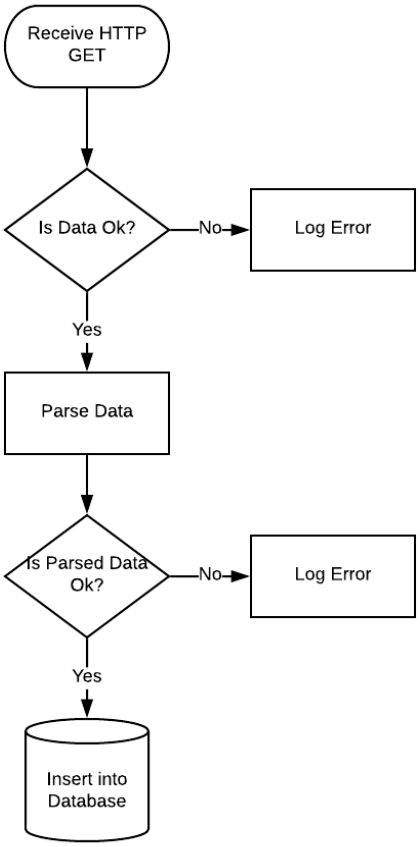

Fig 3.10. Adding data flowchart

3.7. Web Server

This project is going to need means of accessing the mentioned database and also, some kind of interface that allows the user to interact with the data captured by the sensors. In order to fulfill this needs, a web server was developed.

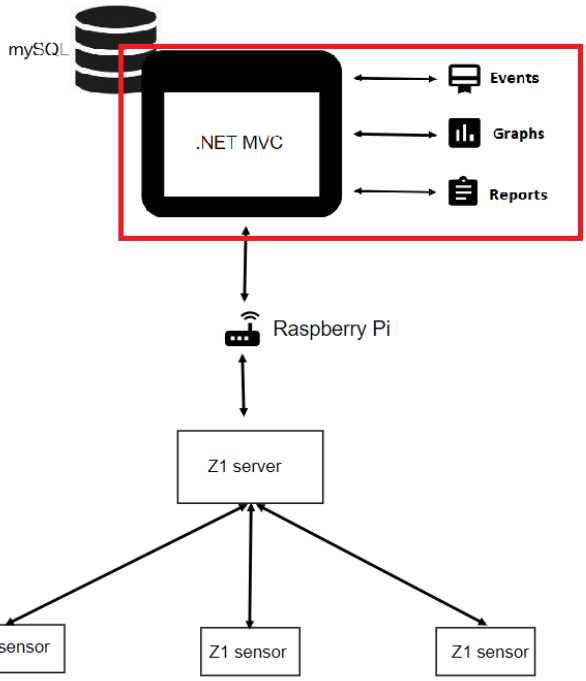

Fig 3.11. Location of the web server inside the whole system

The web server was initially meant to run in the raspberry Pi and be developed using python stack technologies, but after much thought and testing, this idea was deemed not viable and the web server was finally developed using ASP.NET 4.5 technology instead and deployed on a personal computer together with the mySQL Database. This computer then, is made accessible from external users using IIS (Internet Information Services) and configuring the LAN router properly.

This webserver has a set of accessible REST Endpoints that allow the user to interact with the data collected by the WSN and developed according to the MVC (Model-View-Controller) architectural pattern.

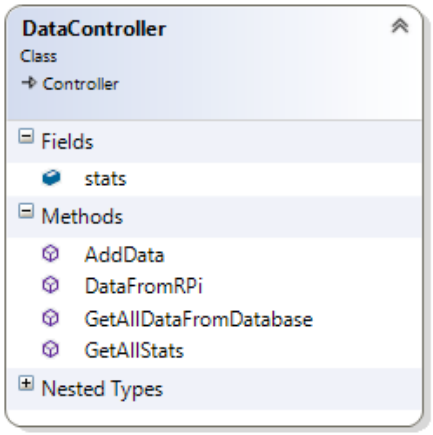

Fig 3.12. Web server REST EndPoints

- **AddData()** is used by a console application used for testing purposes. Calling this function will Add a new register to the database with random values.

- **DataFromRPi(string data)** is the function used to parse the information retrieved by the sensors, collecting its relevant stats and then inserting the values to the database. It's critical that the methods called in this flow are properly optimized is due to the number of times this thread is called on run-time.

```
public string DataFromRPi(string data)
{
    try
    {
        var samples = data.Split('|');
        foreach (string sample in samples)
        {
            if (!string.IsNullOrEmpty(sample))
            {
                //Parse data from raspberry Pi
                var datapoints = sample.Split(';'); //0- disp //1- temp 2- bateria 3- acc

                //Adds relevant stats to a static class saved in server cache.
                BusinessClass.ManageStats(datapoints, stats);

                //Inserts the newly parsed measurements into the database.
                BusinessClass.InsertIntoDB(datapoints);
            }
        }

        return "ok";
    }
    catch (Exception ex)
    {
        BusinessClass.Logger(ex);
        throw ex;
    }
}
```

Code snippet 3.7. DataFromRPI() function

- **GetAllDataFromDatabase()** and **GetAllStats().** This functions are used by the graph and stats utilities and will be properly explained in chapters [3.7.1.] and [3.7.2.]

3.7.1. Graph Utilities

One of the utilities provided by the developed web server is the graph utilities. Using this utilities the user will be ables to see a real-time representation of what the sensing nodes just sensed with a fraction of a second delay and making full use of the overall high-speed design of the network.

The graph interface is pretty simple, the user can select a device to filter data sent by a specific node and also select which kind of measurement he wants to see. After selecting both, an auto-updating graph will appear on the screen with the desired data.

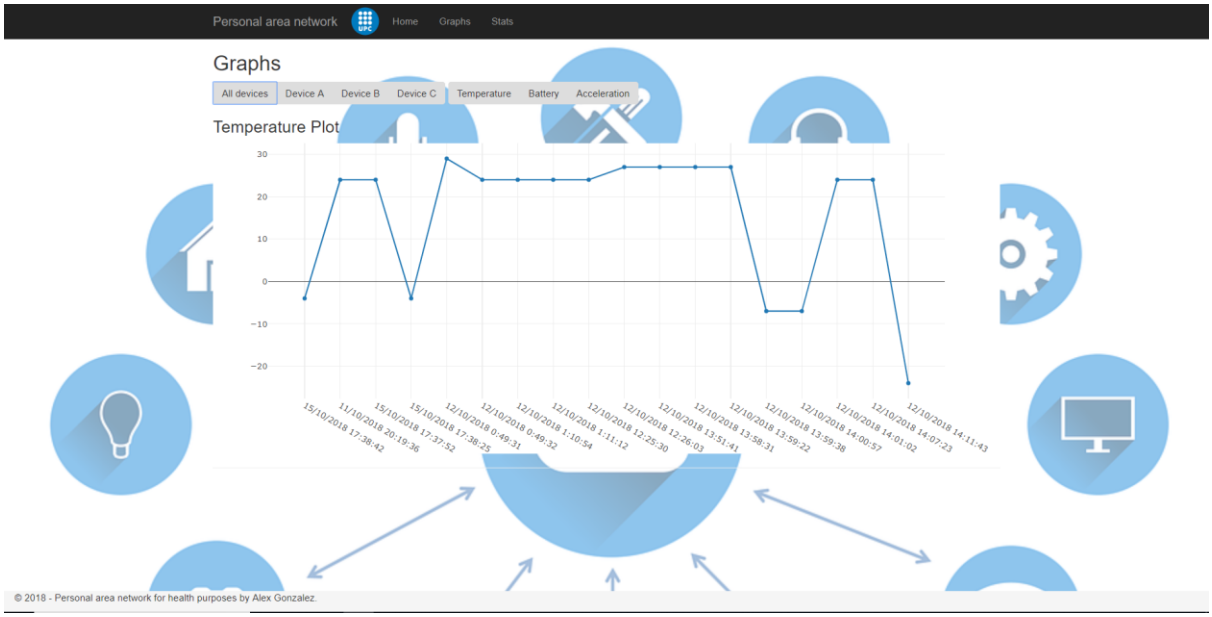

Fig 3.13. Web server graph interface

The information shown in the graph is directly pulled from the database. The data structure itself is rather simple, it consists of four Lists, the first of devices and the others with information related to the device with the same index. This meaning that acceleration[i] is the acceleration sensed by disp[i].

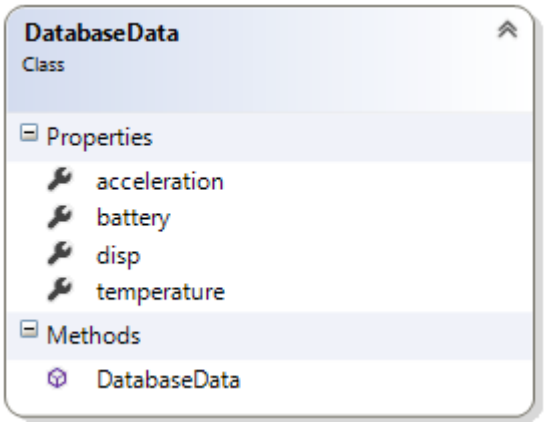

Fig 3.14. Web server graphs data model

In this specific case, since the amount of data gathered by the sensors is so overwhelming for a proper visualization, some limiting factors where to be implemented. After some testing, this was the final query used to pull the data from the mySQL database:

```
SELECT * FROM (SELECT *,ROW_NUMBER() OVER(PARTITION BY timestamp ORDER BY idmeasurements DESC) rn FROM tfg_personalareanetwork.measurements where disp='*deviceId*  order by timestamp desc limit 200) a where rn=1
```

Code snippet 3.8. Retrieve data SQL query

This query gathers the last 200 results that were gathered in different recorded times. Effectively being able to retrieve data from a given device and measurement from the last 200 seconds.

3.7.2. Stats Utilities

Since the graph utility only provide information from the last 200 seconds and in order to keep the graphing interface as low-weight as posible, a new static interface was created. The Stats interface provides the user from relevant statistical information from the whole logging session.

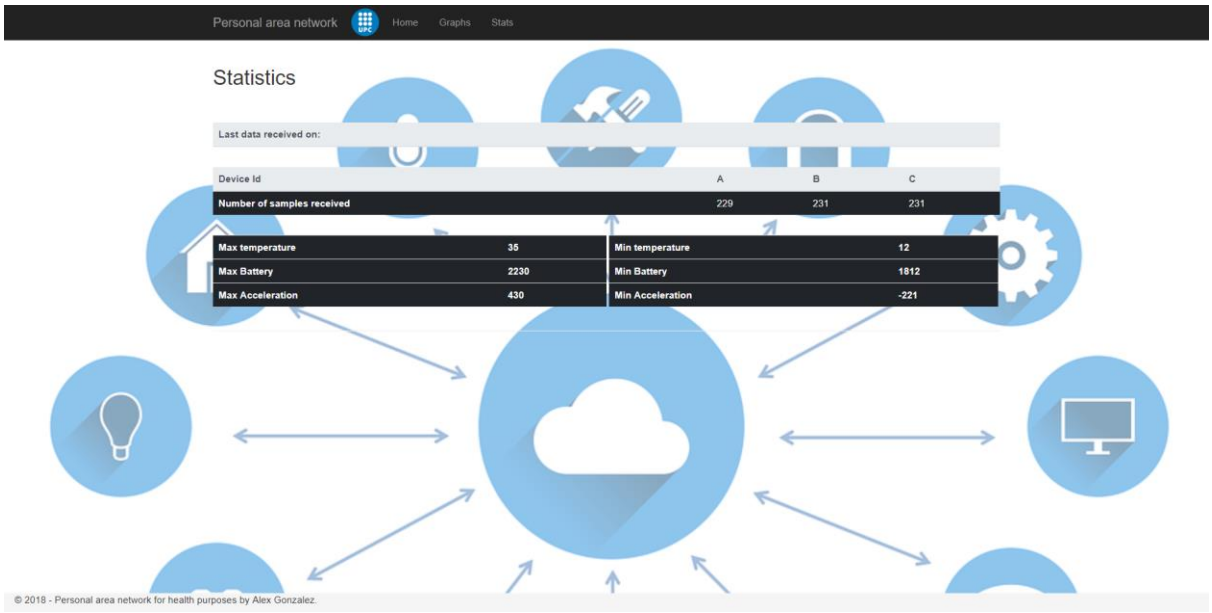

Fig 3.15. Web server stats interface

Contrary from the graphing utility, the stats interface doesn't pull data directly from the Database, instead, the data is saved in server side session cache, enabling the user to reset this data without deleting any relevant historical information from the database simply restarting the server.

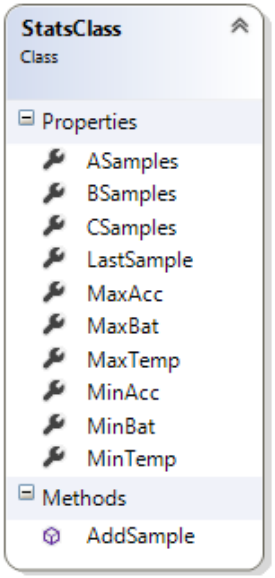

Fig 3.16. Web server stats data model

4. TESTING SCENARIO AND RESULTS

4.1. Connectivity tests

To verify the connectivity in the deployed network, few connectivity tests were done. These tests were design to ensure the correct behaviour of the devices used and the connections between devices.

4.1.1 Web server accessibility test

The purpose of this test was to make sure that the web server was publicly available to final users. In order to do this test, a console application, written in c# was developed.

This console application sends a string of verified valid data to the webserver using an HTTP GET and verifies that the response of the server is an HTTP OK. Then, the Database tables were checked to make sure that the data was correctly inserted. The test was conducted as shown in [Fig .4.1.] flow chart and connection was verified.

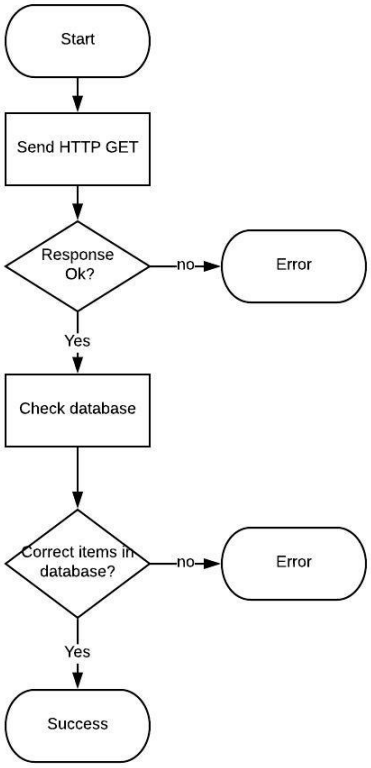

Fig 4.1. Web server accessibility test flow chart

4.1.1. End-to-end connectivity test

This test was performed after every device was properly configured and every code piece was verified to be working as intended. The purpose of this test is to assure that data sensed at the sensor nodes reaches the database and is properly shown on user interfaces.

During the testing, low-speed sensor node software was used, executing a duty cycle every second instead of at contiki's maximum speed. Some default contiki parameters worth mentioning are: the 802.15.4 channel used is channel 26, this channel avoid interference with Wi-Fi; the transmission power is set to 0 dBm (1 mW), which is the maximum allowed by the radio CC2420; MCU operation frequency in active mode is 8MHz, this can be found in contiki3.0/cpu/msp430/f2xxx/msp430.c; and the MCU low power mode is LPM3, this can be found in contiki-3.0/cpu/msp430/f2xxx/msp430.c.

This test was performed using the definite network topography seen in [Fig 3.1.]. After turning on every device, the Database was checked to see if data was being correctly inserted into the designated table. To see if any data package was missing and knowing the frequency at which data was being sensed, the following SQL query was used:

```
SELECT count( *) FROM FROM tfg_personalareanetwork.measurements
```

This query provides the amount of entries for a given table, which should be equal to three times the time that has passed since the last device was turned On:

```
count(*) = 3 * t[s]
```

If that's the case, the test has been passed and connectivity across the network has been assured.

4.2. Speed Test

The purpose of this test is to calculate the maximum sampling frequency possible by the sensors nodes that allows the network to function normally.

As explained at introduction and then calculated in chapter [1.3.], the ideal minimum sampling frequency to be obtained by this network should be higher

than 4KHz. This has been proven not to be posible with the technology stack used in this project. A more specialized environment would be needed to attain those kind of results.

To achieve our goal of maximum network throughput, the etimer library and internal Contiki clocks were modified to allow for a faster ticking clock. To do so a couple Contiki configuration files were edited:

- /platform/z1/contiki-conf.h: I changed the CLOCK_CONF_SECOND to 4096
- /cpu/msp430/clock.c: I deleted the ID_3 & change the INTERVAL var to 64

Increasing the clock resolution even further would result in most of the packets being lost before reaching the UDP server Zolertia Z1 node.

In order to measure the total throughput of the network, a python script was developed. The script reads the USB serial port and logs every time data is received. Once 1000 data packets are received, the time elapsed is calculated. This script will be provided in Annex I and its use can be better understood using the following flowchart [Fig 4.2.]

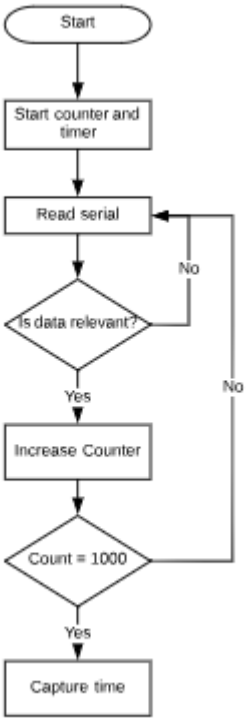

Fig 4.2. Speed test flowchart

This test was run 3 times in arrow to and the median result was taken as definite, the three results obtained were:

- 1st test: 37.012s
- 2nd test: 36.871s
- 3rd test: 36.812s

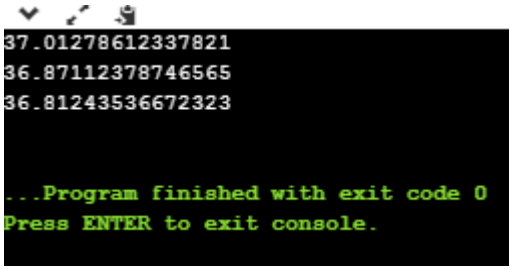

Fig 4.3. Python console after executing the script

The average time of the results is 36.89s. Taking into account that every data packet has six triplets of data -see chapter [3.4.]- we can conclude that the attained sampling frequency is 158.35 Hz. This meaning that a sample of temperature, battery and acceleration is obtained every 6.315 ms.

CONCLUSIONS

This project focuses on the design and implementation of an IoT application using sensor nodes based on the low power microcontroller MSP430. To achieve the objective of this project, a thorough review of the IoT and Internet technologies has been done. This review starts with some basic concepts of IoT and its basic architecture. The review then continues with a description of the technologies that were used on this project as well as technologies that made the Internet of Things possible in the first place.

Throughout the development of this project, energy consumption has not been a major concern. The aim of this project has been to provide the user with a high-speed personal area network network rather than an energy-efficient one, and since it's main application is implementing a WBAN, an energy source could be always at reach.

A high-speed WBAN has several challenges to overcome. The main difficulty of this project has been precisely the high-speed requirements. The state of the art technologies used in this project had been design to be energy-efficient and reliable at low-speeds. The whole development process turns increasingly difficult as your network speed needs increase.

In the end, a sampling frequency of 158 Hz was obtained, which is far away from the 4KHz needed to be able to sample a phonocardiogram. In order to do so, a more specialized device with high-end data compression capabilities would have to be used.

In this work security aspects were not addressed since the data collected by the network is not susceptible to be stolen. Using the security capabilities of the IEEE 802.15.4 standard would result in the decrease of network speed and its data transmission capabilities. Future works could evaluate the trade-offs between speed efficiency and security. In this line Contiki 3.0 can provide link-layer encryption for IEEE 802.15.4 radio.

Finally, it is also worth mentioning, that all the applications and software developed for this project can be used in another deployments. The programs developed for the nodes are hardware independent and can be used for other hardware platforms running ContikiOS.

ACRONYMS

6LoWPAN	IPV6 Over Low Power Wireless Personal Area Networks
ADC	Analog to Digital Converter
ASP	Active Server Pages
CPU	Central Processing Unit
DAG	Directed Acyclic Graphs
DAO	Destination Advertisement Object
DHCP	Dynamic Host Configuration Protocol
DIO	DODAG Information Object
DIS	DODAG Information Solicitation
DODAG	Destination Oriented DAG
DSI	Display Interface
EU	European Union
GPIO	General Purpose Input/Output
GPU	Graphics Processor Unit
HTTP	Hypertext Transfer Protocol
ICTs	Information and Communication Technologies
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IIS	Internet Information Services
IoT	Internet of Things
IP	Internet Protocol
ITU-T	ITU Telecommunication Standardization Sector
LAN	Local Area Network
LPM	Low Power Mode
MAC	Media Access Control
MCU	Microcontroller Unit
MP2P	Multipoint-to-Point
MTU	Maximum Transmission Unit
MVC	Model-View-Controller
NAT	Network Address Translation
NDP	Neighbor Discovery Protocol
P2MP	Point-to-Multipoint
P2P	Point-to-Point
PAN	Personal Area Network
QoS	Quality of Service
RA	Router Advertisement
RAM	Random Access Memory
RDC	Radio Duty Cycle

RF	Radio Frequency
RPL	Routing Protocol for Low-Power and Lossy Networks
SBC	Single Board Computer
SPI	Serial Peripheral Interface
TCP	Transmission Control Protocol
TDMA	Time Division Multiple Access
UART	Universal Asynchronous Receiver/Transmitter
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
USB	Universal Serial Bus
VM	Virtual Machine
WPAN	Wireless Personal Area Network
WSN	Wireless Sensor Network

REFERENCES

[1] https://avoa.com/2016/07/05/understanding-the-five-tiers-of-iot-core-architecture/ March, 2018.

[2] Linan, A, Vives, A, Bagula, A, Zennaro, M, Pietrosemoli, E, IoT in five Days, 2016.

[3] https://www.internetsociety.org/deploy360/ipv6/ March, 2018

[4] https://www.webopedia.com/DidYouKnow/Internet/ipv6_ipv4_ difference.html March, 2018

[5] https://www.google.com/intl/en/ipv6/statistics.html October, 2018

[6] https://www.link-labs.com/blog/6lowpan-vs-zigbee September, 2018

[7] [RFC4944] http://www.rfc-editor.org/rfc/pdfrfc/rfc4944.txt.pdf September, 2018

[8] https://www.researchgate.net/figure/IP-and-6LoWPAN-protocol-stack-in-reference-to-layers -of-the-TCP-IP-networking-model_fig1_281333084 October, 2018

[9] https://tools.ietf.org/html/draft-daniel-6lowpan-security-analysis-05 October, 2018

[10] https://www.embedded.com/electronics-blogs/embedded-cloud-talkers /4236873/How-to-setup-a-6LoWPAN-network October, 2018

[11] International Telecommunication Union, Recommendation ITU-T Y.2060: Overview of the Internet of things, 2012.

[12] http://www.contiki-os.org/ October, 2018

[13] http://eprints.ugd.edu.mk/16096/1/Zbornik-ITRO-2016-283-289.pdf

[14] https://inrg.soe.ucsc.edu/howto-setup-instant-contiki-with-virtualbox/ October, 2018

[15] Zolertia Z1 Datasheet http://zolertia.sourceforge.net/wiki/images/e/e8/

Z1_RevC_Datasheet.pdf October, 2018

[16] https://github.com/Zolertia/Resources/wiki/The-Z1-mote October, 2018

[17] https://en.wikipedia.org/wiki/Raspberry_Pi October, 2018

[18] https://www.sparkfun.com/products/retired/11546 October, 2018

[19] https://github.com/raspberrypi/documentation October, 2018

[20] Ahmad Ali 1, Yu Ming , Sagnik Chakraborty and Saima Iram. A Comprehensive Survey on Real-Time Applications of WSN, 2017.
Available here: https://www.mdpi.com/1999-5903/9/4/77

[21] [RFC6550] RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks, 2012.
Available here: https://tools.ietf.org/html/rfc6550

[22] https://www.ncbi.nlm.nih.gov/pubmed/26871603

[23] https://www.raspberrypi.org/downloads/raspbian/

[24] Mohamed Marwan Selim. An Empirical Analysis on the Microservices Architecture Pattern and Service-Oriented Architecture, 2016.
Available here: https://www.researchgate.net/publication/313113813_An_Empirical_Analysis_on_the_Microservices_Architecture_Pattern_and_Service-Oriented_Architecture

[25] Afonso Oliveira, Teresa Vazão. Low-power and lossy networks under mobility: A survey, 2016
Available here: https://www.sciencedirect.com/science/article/abs/pii/S1389128616300895

ANNEX I. PROGRAMS USED

Code I. Python script running at Raspberry Pi.

```
import serial, string, requests, time
counter = 0
total_strings = ""
output =" "

#Packet blocks to send to the server
blocks_per_packet=25

# IP Address of the web server
ip_webserver ="192.168.1.110"

ser = serial.Serial('/dev/ttyUSB0',115200, 8, 'N', 1, timeout=1)
if ser != null:
    while True:
        print ("----")
        while output != "":
            output =ser.readline().decode("utf-8")
            if len(output) > 5 :
                counter=counter+1
                total_strings+=output
                if (counter>=blocks_per_packet):
                    counter = 0
                    r =
requests.get(("http://192.168.1.110/data/DataFromRPI?data="+total_strings).replace("\n","").replace("\r",""))
                    total_strings=""
            output = " "
else print("No serial device Found")
r = requests.get("http://192.168.1.110/data/Logger?data=RPI Border router error: No serial device found");
```

Code II. C script running at Zolertia Z1 sensor node A.

```
#include "contiki.h"
#include "lib/random.h"
#include "sys/ctimer.h"
#include "net/uip.h"
#include "net/uip-ds6.h"
#include "net/uip-udp-packet.h"
#include "sys/ctimer.h"
#include <stdio.h>
#include <string.h>
#include "dev/i2cmaster.h"  // Include IC driver
#include "dev/battery-sensor.h"
#include "dev/tmp102.h"     // Include sensor driver
#include "dev/adxl345.h"
#define UDP_CLIENT_PORT 8765
#define UDP_SERVER_PORT 5678
#define UDP_EXAMPLE_ID  190
#define DEBUG DEBUG_PRINT
#include "net/uip-debug.h"

#ifndef PERIOD
#define PERIOD 1
#endif

#define START_INTERVAL              (1)
#define SEND_INTERVAL               (PERIOD * 1)
#define SEND_TIME                   (random_rand() % (SEND_INTERVAL))
#define MAX_PAYLOAD_LEN                     108
```

```
#define SAMPLES_PER_PACKAGE 6
static struct uip_udp_conn *client_conn;
static uip_ipaddr_t server_ipaddr;
static uint16_t counter = 0;
static char temp[MAX_PAYLOAD_LEN/SAMPLES_PER_PACKAGE];
static char payload[MAX_PAYLOAD_LEN];
int16_t z, tempint, raw;
uint16_t absraw;
uint16_t bateria;
float mv;
/*---------------------------------------------------------------------------*/
PROCESS(udp_client_process, "UDP client process");
AUTOSTART_PROCESSES(&udp_client_process);
/*---------------------------------------------------------------------------*/
static void
tcpip_handler(void)
{
  char *str;

  if(uip_newdata()) {
    str = uip_appdata;
    str[uip_datalen()] = '\0';
    printf("DATA recv '%s'\n", str);
  }
}
/*---------------------------------------------------------------------------*/
static void send_packet(void)
{
  static int seq_id;
  char buf[MAX_PAYLOAD_LEN] ;
  //Copy buffer to a temporal variable to send
  strcpy(buf,payload);
  //Restart packet counter
  counter = 0;
  //Restart buffer
  strcpy(payload,"");
  //In case you want to track packet numbers
  seq_id++;
  //Send Data
  uip_udp_packet_sendto(client_conn, buf, strlen(buf),
                        &server_ipaddr, UIP_HTONS(UDP_SERVER_PORT));
}
/*---------------------------------------------------------------------------*/
/* This function read values the sensors and stores the data on a buffer*/

static void get_and_save_data(void)
{
  uint32_t aux;

  if(payload != NULL) {
                //Measure battery
              bateria = battery_sensor.value(0);
              mv = (bateria * 2.50 * 2) / 4096;

    //Measure Temperature
              raw = tmp102_read_temp_raw();
    absraw = raw;
    if (raw < 0) {
      absraw = (raw ^ 0xFFFF) + 1;
    }
              tempint  = (absraw >> 8) ;

    //Measure Acceleration
              z_acc = accm_read_axis(Z_AXIS);

    //Save Data into Buffer
              sprintf(temp, "a;%d;%i;%d|",  tempint, bateria, z_acc);
              strcat(payload, temp);
              counter++;               //Add 1 to number of sample packets in  buffer
  }
  else {printf("No memory available");}
}
/*---------------------------------------------------------------------------*/
```

```
static void
print_local_addresses(void)
{
  int i;
  uint8_t state;

  PRINTF("Client IPv6 addresses: ");
  for(i = 0; i < UIP_DS6_ADDR_NB; i++) {
    state = uip_ds6_if.addr_list[i].state;
    if(uip_ds6_if.addr_list[i].isused &&
       (state == ADDR_TENTATIVE || state == ADDR_PREFERRED)) {
      PRINT6ADDR(&uip_ds6_if.addr_list[i].ipaddr);
      PRINTF("\n");
      /* hack to make address "final" */
      if (state == ADDR_TENTATIVE) {
            uip_ds6_if.addr_list[i].state = ADDR_PREFERRED;
      }
    }
  }
}
/*---------------------------------------------------------------------------*/
static void
set_global_address(void)
{
  uip_ipaddr_t ipaddr;

  uip_ip6addr(&ipaddr, 0xaaaa, 0, 0, 0, 0, 0, 0, 0);
  uip_ds6_set_addr_iid(&ipaddr, &uip_lladdr);
  uip_ds6_addr_add(&ipaddr, 0, ADDR_AUTOCONF);

#if 0
/* Mode 1 - 64 bits inline */
   uip_ip6addr(&server_ipaddr, 0xaaaa, 0, 0, 0, 0, 0, 0, 1);
#elif 1
/* Mode 2 - 16 bits inline */
  uip_ip6addr(&server_ipaddr, 0xaaaa, 0, 0, 0, 0, 0x00ff, 0xfe00, 1);
#else
/* Mode 3 - derived from server link-local (MAC) address */
  uip_ip6addr(&server_ipaddr, 0xaaaa, 0, 0, 0, 0x0250, 0xc2ff, 0xfea8, 0xcd1a); //redbee-econotag
#endif
}
/*---------------------------------------------------------------------------*/
PROCESS_THREAD(udp_client_process, ev, data)
{
  static struct etimer periodic;
  static struct ctimer backoff_timer;
#if WITH_COMPOWER
  static int print = 0;
#endif

  PROCESS_BEGIN();

  PROCESS_PAUSE();

  set_global_address();
                tmp102_init();
    SENSORS_ACTIVATE(battery_sensor);
              accm_init();

  PRINTF("UDP client process started\n");

  print_local_addresses();

  /* new connection with remote host */
  client_conn = udp_new(NULL, UIP_HTONS(UDP_SERVER_PORT), NULL);
  if(client_conn == NULL) {
    PRINTF("No UDP connection available, exiting the process!\n");
    PROCESS_EXIT();
  }
  udp_bind(client_conn, UIP_HTONS(UDP_CLIENT_PORT));

  PRINTF("Created a connection with the server ");
  PRINT6ADDR(&client_conn->ripaddr);
  PRINTF(" local/remote port %u/%u\n",
```

```
            UIP_HTONS(client_conn->lport), UIP_HTONS(client_conn->rport));

  etimer_set(&periodic, SEND_INTERVAL);

  while(1) {
    PROCESS_YIELD();
    if(ev == tcpip_event) {
      tcpip_handler();
    }

    get_and_save_data();

    if(etimer_expired(&periodic)) {
      etimer_reset(&periodic);

            if (counter == SAMPLES_PER_PACKAGE){
              ctimer_set(&backoff_timer, SEND_TIME, send_packet, NULL);
     }
    }
  }

  PROCESS_END();
}
/*---------------------------------------------------------------------------*/
```

Code III. C script running at Zolertia Z1 sensor node B.

```
#include "contiki.h"
#include "lib/random.h"
#include "sys/ctimer.h"
#include "net/uip.h"
#include "net/uip-ds6.h"
#include "net/uip-udp-packet.h"
#include "sys/ctimer.h"
#include <stdio.h>
#include <string.h>
#include "dev/i2cmaster.h"  // Include IC driver
#include "dev/battery-sensor.h"
#include "dev/tmp102.h"     // Include sensor driver
#include "dev/adxl345.h"
#define UDP_CLIENT_PORT 8765
#define UDP_SERVER_PORT 5678
#define UDP_EXAMPLE_ID  190
#define DEBUG DEBUG_PRINT
#include "net/uip-debug.h"

#ifndef PERIOD
#define PERIOD 1
#endif

#define START_INTERVAL                (1)
#define SEND_INTERVAL                 (PERIOD * 1)
#define SEND_TIME                     (random_rand() % (SEND_INTERVAL))
#define MAX_PAYLOAD_LEN                     108
#define SAMPLES_PER_PACKAGE 6
static struct uip_udp_conn *client_conn;
static uip_ipaddr_t server_ipaddr;
static uint16_t counter = 0;
static char temp[MAX_PAYLOAD_LEN/SAMPLES_PER_PACKAGE];
static char payload[MAX_PAYLOAD_LEN];
int16_t z, tempint, raw;
uint16_t absraw;
uint16_t bateria;
float mv;
/*---------------------------------------------------------------------------*/
PROCESS(udp_client_process, "UDP client process");
AUTOSTART_PROCESSES(&udp_client_process);
/*---------------------------------------------------------------------------*/
static void
```

```
tcpip_handler(void)
{
  char *str;

  if(uip_newdata()) {
    str = uip_appdata;
    str[uip_datalen()] = '\0';
    printf("DATA recv '%s'\n", str);
  }
}
/*---------------------------------------------------------------------------*/
static void send_packet(void)
{
  static int seq_id;
  char buf[MAX_PAYLOAD_LEN] ;
  //Copy buffer to a temporal variable to send
  strcpy(buf,payload);
  //Restart packet counter
  counter = 0;
  //Restart buffer
  strcpy(payload,"");
  //In case you want to track packet numbers
  seq_id++;
  //Send Data
  uip_udp_packet_sendto(client_conn, buf, strlen(buf),
                        &server_ipaddr, UIP_HTONS(UDP_SERVER_PORT));
}
/*---------------------------------------------------------------------------*/
/* This function read values the sensors and stores the data on a buffer*/

static void get_and_save_data(void)
{
  uint32_t aux;

  if(payload != NULL) {
                  //Measure battery
                bateria = battery_sensor.value(0);
                mv = (bateria * 2.50 * 2) / 4096;

    //Measure Temperature
                raw = tmp102_read_temp_raw();
    absraw = raw;
    if (raw < 0) {
      absraw = (raw ^ 0xFFFF) + 1;
    }
                tempint  = (absraw >> 8) ;

    //Measure Acceleration
                z_acc = accm_read_axis(Z_AXIS);

    //Save Data into Buffer
                sprintf(temp, "b;%d;%i;%d|",  tempint, bateria, z_acc);
                strcat(payload, temp);
                counter++;              //Add 1 to number of sample packets in  buffer
  }
  else {printf("No memory available");}
}
/*---------------------------------------------------------------------------*/

static void
print_local_addresses(void)
{
  int i;
  uint8_t state;

  PRINTF("Client IPv6 addresses: ");
  for(i = 0; i < UIP_DS6_ADDR_NB; i++) {
    state = uip_ds6_if.addr_list[i].state;
    if(uip_ds6_if.addr_list[i].isused &&
       (state == ADDR_TENTATIVE || state == ADDR_PREFERRED)) {
      PRINT6ADDR(&uip_ds6_if.addr_list[i].ipaddr);
      PRINTF("\n");
      /* hack to make address "final" */
      if (state == ADDR_TENTATIVE) {
```

```
            uip_ds6_if.addr_list[i].state = ADDR_PREFERRED;
      }
    }
  }
}
/*---------------------------------------------------------------------------*/
static void
set_global_address(void)
{
  uip_ipaddr_t ipaddr;

  uip_ip6addr(&ipaddr, 0xaaaa, 0, 0, 0, 0, 0, 0, 0);
  uip_ds6_set_addr_iid(&ipaddr, &uip_lladdr);
  uip_ds6_addr_add(&ipaddr, 0, ADDR_AUTOCONF);

#if 0
/* Mode 1 - 64 bits inline */
   uip_ip6addr(&server_ipaddr, 0xaaaa, 0, 0, 0, 0, 0, 0, 1);
#elif 1
/* Mode 2 - 16 bits inline */
  uip_ip6addr(&server_ipaddr, 0xaaaa, 0, 0, 0, 0, 0x00ff, 0xfe00, 1);
#else
/* Mode 3 - derived from server link-local (MAC) address */
  uip_ip6addr(&server_ipaddr, 0xaaaa, 0, 0, 0, 0x0250, 0xc2ff, 0xfea8, 0xcd1a); //redbee-econotag
#endif
}
/*---------------------------------------------------------------------------*/
PROCESS_THREAD(udp_client_process, ev, data)
{
  static struct etimer periodic;
  static struct ctimer backoff_timer;
#if WITH_COMPOWER
  static int print = 0;
#endif

  PROCESS_BEGIN();

  PROCESS_PAUSE();

  set_global_address();
                tmp102_init();
    SENSORS_ACTIVATE(battery_sensor);
            accm_init();

  PRINTF("UDP client process started\n");

  print_local_addresses();

  /* new connection with remote host */
  client_conn = udp_new(NULL, UIP_HTONS(UDP_SERVER_PORT), NULL);
  if(client_conn == NULL) {
    PRINTF("No UDP connection available, exiting the process!\n");
    PROCESS_EXIT();
  }
  udp_bind(client_conn, UIP_HTONS(UDP_CLIENT_PORT));

  PRINTF("Created a connection with the server ");
  PRINT6ADDR(&client_conn->ripaddr);
  PRINTF(" local/remote port %u/%u\n",
            UIP_HTONS(client_conn->lport), UIP_HTONS(client_conn->rport));

  etimer_set(&periodic, SEND_INTERVAL);

  while(1) {
    PROCESS_YIELD();
    if(ev == tcpip_event) {
      tcpip_handler();
    }

    get_and_save_data();

    if(etimer_expired(&periodic)) {
      etimer_reset(&periodic);
```

```
            if (counter == SAMPLES_PER_PACKAGE){
              ctimer_set(&backoff_timer, SEND_TIME, send_packet, NULL);
     }
    }
  }

  PROCESS_END();
}
/*---------------------------------------------------------------------------*/
```

Code IV. C script running at Zolertia Z1 sensor node C.

```
#include "contiki.h"
#include "lib/random.h"
#include "sys/ctimer.h"
#include "net/uip.h"
#include "net/uip-ds6.h"
#include "net/uip-udp-packet.h"
#include "sys/ctimer.h"
#include <stdio.h>
#include <string.h>
#include "dev/i2cmaster.h"  // Include IC driver
#include "dev/battery-sensor.h"
#include "dev/tmp102.h"     // Include sensor driver
#include "dev/adxl345.h"
#define UDP_CLIENT_PORT 8765
#define UDP_SERVER_PORT 5678
#define UDP_EXAMPLE_ID  190
#define DEBUG DEBUG_PRINT
#include "net/uip-debug.h"

#ifndef PERIOD
#define PERIOD 1
#endif

#define START_INTERVAL                (1)
#define SEND_INTERVAL                 (PERIOD * 1)
#define SEND_TIME                     (random_rand() % (SEND_INTERVAL))
#define MAX_PAYLOAD_LEN                     108
#define SAMPLES_PER_PACKAGE 6
static struct uip_udp_conn *client_conn;
static uip_ipaddr_t server_ipaddr;
static uint16_t counter = 0;
static char temp[MAX_PAYLOAD_LEN/SAMPLES_PER_PACKAGE];
static char payload[MAX_PAYLOAD_LEN];
int16_t z, tempint, raw;
uint16_t absraw;
uint16_t bateria;
float mv;
/*---------------------------------------------------------------------------*/
PROCESS(udp_client_process, "UDP client process");
AUTOSTART_PROCESSES(&udp_client_process);
/*---------------------------------------------------------------------------*/
static void
tcpip_handler(void)
{
  char *str;

  if(uip_newdata()) {
    str = uip_appdata;
    str[uip_datalen()] = '\0';
    printf("DATA recv '%s'\n", str);
  }
}
/*---------------------------------------------------------------------------*/
static void send_packet(void)
{
  static int seq_id;
  char buf[MAX_PAYLOAD_LEN] ;
```

```
  //Copy buffer to a temporal variable to send
  strcpy(buf,payload);
  //Restart packet counter
  counter = 0;
  //Restart buffer
  strcpy(payload,"");
  //In case you want to track packet numbers
  seq_id++;
  //Send Data
  uip_udp_packet_sendto(client_conn, buf, strlen(buf),
                        &server_ipaddr, UIP_HTONS(UDP_SERVER_PORT));
}
/*---------------------------------------------------------------------------*/
/* This function read values the sensors and stores the data on a buffer*/

static void get_and_save_data(void)
{
  uint32_t aux;

  if(payload != NULL) {
                //Measure battery
               bateria = battery_sensor.value(0);
               mv = (bateria * 2.50 * 2) / 4096;

    //Measure Temperature
               raw = tmp102_read_temp_raw();
    absraw = raw;
    if (raw < 0) {
      absraw = (raw ^ 0xFFFF) + 1;
    }
               tempint  = (absraw >> 8) ;

    //Measure Acceleration
               z_acc = accm_read_axis(Z_AXIS);

    //Save Data into Buffer
               sprintf(temp, "c;%d;%i;%d|",  tempint, bateria, z_acc);
               strcat(payload, temp);
               counter++;               //Add 1 to number of sample packets in  buffer
  }
  else {printf("No memory available");}
}
/*---------------------------------------------------------------------------*/

static void
print_local_addresses(void)
{
  int i;
  uint8_t state;

  PRINTF("Client IPv6 addresses: ");
  for(i = 0; i < UIP_DS6_ADDR_NB; i++) {
    state = uip_ds6_if.addr_list[i].state;
    if(uip_ds6_if.addr_list[i].isused &&
       (state == ADDR_TENTATIVE || state == ADDR_PREFERRED)) {
      PRINT6ADDR(&uip_ds6_if.addr_list[i].ipaddr);
      PRINTF("\n");
      /* hack to make address "final" */
      if (state == ADDR_TENTATIVE) {
               uip_ds6_if.addr_list[i].state = ADDR_PREFERRED;
      }
    }
  }
}
/*---------------------------------------------------------------------------*/
static void
set_global_address(void)
{
  uip_ipaddr_t ipaddr;

  uip_ip6addr(&ipaddr, 0xaaaa, 0, 0, 0, 0, 0, 0, 0);
  uip_ds6_set_addr_iid(&ipaddr, &uip_lladdr);
  uip_ds6_addr_add(&ipaddr, 0, ADDR_AUTOCONF);
```

```
#if 0
/* Mode 1 - 64 bits inline */
   uip_ip6addr(&server_ipaddr, 0xaaaa, 0, 0, 0, 0, 0, 1);
#elif 1
/* Mode 2 - 16 bits inline */
  uip_ip6addr(&server_ipaddr, 0xaaaa, 0, 0, 0, 0, 0x00ff, 0xfe00, 1);
#else
/* Mode 3 - derived from server link-local (MAC) address */
  uip_ip6addr(&server_ipaddr, 0xaaaa, 0, 0, 0, 0x0250, 0xc2ff, 0xfea8, 0xcd1a); //redbee-econotag
#endif
}
/*---------------------------------------------------------------------------*/
PROCESS_THREAD(udp_client_process, ev, data)
{
  static struct etimer periodic;
  static struct ctimer backoff_timer;
#if WITH_COMPOWER
  static int print = 0;
#endif

  PROCESS_BEGIN();

  PROCESS_PAUSE();

  set_global_address();
                tmp102_init();
    SENSORS_ACTIVATE(battery_sensor);
              accm_init();

  PRINTF("UDP client process started\n");

  print_local_addresses();

  /* new connection with remote host */
  client_conn = udp_new(NULL, UIP_HTONS(UDP_SERVER_PORT), NULL);
  if(client_conn == NULL) {
    PRINTF("No UDP connection available, exiting the process!\n");
    PROCESS_EXIT();
  }
  udp_bind(client_conn, UIP_HTONS(UDP_CLIENT_PORT));

  PRINTF("Created a connection with the server ");
  PRINT6ADDR(&client_conn->ripaddr);
  PRINTF(" local/remote port %u/%u\n",
              UIP_HTONS(client_conn->lport), UIP_HTONS(client_conn->rport));

  etimer_set(&periodic, SEND_INTERVAL);

  while(1) {
    PROCESS_YIELD();
    if(ev == tcpip_event) {
      tcpip_handler();
    }

    get_and_save_data();

    if(etimer_expired(&periodic)) {
      etimer_reset(&periodic);

              if (counter == SAMPLES_PER_PACKAGE){
                 ctimer_set(&backoff_timer, SEND_TIME, send_packet, NULL);
      }
    }
  }

  PROCESS_END();
}
/*---------------------------------------------------------------------------*/
```

Code V. C script running at Zolertia Z1 server node.

```
#include "contiki.h"
#include "contiki-lib.h"
#include "contiki-net.h"
#include "net/uip.h"
#include "net/rpl/rpl.h"
#include "net/netstack.h"
#include "dev/button-sensor.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#define DEBUG DEBUG_PRINT
#include "net/uip-debug.h"
#define UIP_IP_BUF   ((struct uip_ip_hdr *)&uip_buf[UIP_LLH_LEN])
#define UDP_CLIENT_PORT 8765
#define UDP_SERVER_PORT         5678
#define UDP_EXAMPLE_ID  190

static struct uip_udp_conn *server_conn;

PROCESS(udp_server_process, "UDP server process");
AUTOSTART_PROCESSES(&udp_server_process);
/*---------------------------------------------------------------------------*/
static void
tcpip_handler(void)
{
  char *appdata;
  //If new data received
  if(uip_newdata()) {
    appdata = (char *)uip_appdata;
    appdata[uip_datalen()] = 0;
    //Send data through serial.
    PRINTF("%s", appdata);
    PRINTF("\n");
  }
}
/*---------------------------------------------------------------------------*/
static void
print_local_addresses(void)
{
  int i;
  uint8_t state;

  PRINTF("Server IPv6 addresses: ");
  for(i = 0; i < UIP_DS6_ADDR_NB; i++) {
    state = uip_ds6_if.addr_list[i].state;
    if(state == ADDR_TENTATIVE || state == ADDR_PREFERRED) {
      PRINT6ADDR(&uip_ds6_if.addr_list[i].ipaddr);
      PRINTF("\n");
      /* hack to make address "final" */
      if (state == ADDR_TENTATIVE) {
            uip_ds6_if.addr_list[i].state = ADDR_PREFERRED;
      }
    }
  }
}
/*---------------------------------------------------------------------------*/
PROCESS_THREAD(udp_server_process, ev, data)
{
  uip_ipaddr_t ipaddr;
  struct uip_ds6_addr *root_if;

  PROCESS_BEGIN();

  PROCESS_PAUSE();

  SENSORS_ACTIVATE(button_sensor);

  PRINTF("UDP server started\n");

#if UIP_CONF_ROUTER
```

```
/* The choice of server address determines its 6LoPAN header compression.
 * Obviously the choice made here must also be selected in udp-client.c.
 *
 * For correct Wireshark decoding using a sniffer, add the /64 prefix to the 6LowPAN protocol preferences,
 * e.g. set Context 0 to aaaa::.  At present Wireshark copies Context/128 and then overwrites it.
 * (Setting Context 0 to aaaa::1111:2222:3333:4444 will report a 16 bit compressed address of
aaaa::1111:22ff:fe33:xxxx)
 * Note Wireshark's IPCMV6 checksum verification depends on the correct uncompressed addresses.
 */

#if 0
/* Mode 1 - 64 bits inline */
   uip_ip6addr(&ipaddr, 0xaaaa, 0, 0, 0, 0, 0, 0, 1);
#elif 1
/* Mode 2 - 16 bits inline */
  uip_ip6addr(&ipaddr, 0xaaaa, 0, 0, 0, 0, 0x00ff, 0xfe00, 1);
#else
/* Mode 3 - derived from link local (MAC) address */
  uip_ip6addr(&ipaddr, 0xaaaa, 0, 0, 0, 0, 0, 0, 0);
  uip_ds6_set_addr_iid(&ipaddr, &uip_lladdr);
#endif

  uip_ds6_addr_add(&ipaddr, 0, ADDR_MANUAL);
  root_if = uip_ds6_addr_lookup(&ipaddr);
  if(root_if != NULL) {
    rpl_dag_t *dag;
    dag = rpl_set_root(RPL_DEFAULT_INSTANCE,(uip_ip6addr_t *)&ipaddr);
    uip_ip6addr(&ipaddr, 0xaaaa, 0, 0, 0, 0, 0, 0, 0);
    rpl_set_prefix(dag, &ipaddr, 64);
    PRINTF("created a new RPL dag\n");
  } else {
    PRINTF("failed to create a new RPL DAG\n");
  }
#endif /* UIP_CONF_ROUTER */

  print_local_addresses();

  /* The data sink runs with a 100% duty cycle in order to ensure high
     packet reception rates. */
  NETSTACK_MAC.off(1);

  server_conn = udp_new(NULL, UIP_HTONS(UDP_CLIENT_PORT), NULL);
  if(server_conn == NULL) {
    PRINTF("No UDP connection available, exiting the process!\n");
    PROCESS_EXIT();
  }
  udp_bind(server_conn, UIP_HTONS(UDP_SERVER_PORT));

  PRINTF("Created a server connection with remote address ");
  PRINT6ADDR(&server_conn->ripaddr);
  PRINTF(" local/remote port %u/%u\n", UIP_HTONS(server_conn->lport),
         UIP_HTONS(server_conn->rport));

  while(1) {
    PROCESS_YIELD();
    if(ev == tcpip_event) {
      tcpip_handler();
    } else if (ev == sensors_event && data == &button_sensor) {
      PRINTF("Initiaing global repair\n");
      rpl_repair_root(RPL_DEFAULT_INSTANCE);
    }
  }

  PROCESS_END();
}
/*---------------------------------------------------------------------------*/
```

Code VI. Python script used to calculate WSN throughput.

```
import serial, string, requests, time
counter = 0
output =" "
blocks_per_packet=100
ser = serial.Serial('/dev/ttyUSB0',115200, 8, 'N', 1, timeout=1)
t0=time.time()
while True:
    print ("----")
    while output != "":
        output =ser.readline().decode("utf-8")
        if len(output) > 5 :
            counter=counter+1
            if (counter>=blocks_per_packet):
                t1=time.time()
                total=t1-t0
                print(total)
                counter = 0
        output = " "
```

Code VII. Shell script used to run the Raspberry Pi Python script at launch.

```
#!/bin/sh
sleep 5
sudo python border-router-script.py
```